2011-09

A quantitative model for assessing visual simulation software architecture

Harder, Robert W.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

A QUANTITATIVE MODEL FOR ASSESSING VISUAL SIMULATION SOFTWARE ARCHITECTURE

by

Robert W. Harder

September 2011

Dissertation Supervisor: Rudolph Darken

This dissertation was done at the MOVES Institute
Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2011	Dissertation

4. TITLE AND SUBTITLE: A Quantitative Model for Assessing Visual Simulation Software Architecture

5. FUNDING NUMBERS

6. AUTHOR(S): Robert W. Harder

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: NA

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)
The U.S. military is the largest single user of simulation in the world, and our visual simulations can be software-intensive systems with a lifespan of many years. Managers of these simulations need tools to help them make better decisions at the architectural level. Currently, no such quantitative models with supporting metrics exist for this purpose. There are properties that are held as positive characteristics in visual simulation architectures. Visual simulation architectures can be distinguished from one another based on three characteristics: (1) openness, as defined by the use of standards, licensing, and support of innovation; (2) reuse, as defined by the potential of being used in subsequent projects; and (3) agility, as defined by the ease with which software can be integrated, reconfigured, or repurposed. In this research, we propose quantifiable models to measure openness, reuse, and agility, and claim that the models adequately distinguish visual simulation frameworks from one another. Furthermore, we claim that these models can enhance military acquisition decisions. The results show that application of the metrics offers a level of granularity that is useful in identifying key differences in simulation frameworks that could have profound downstream implications.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software metrics, visual simulation, architecture, frameworks, openness, reuse, reusability, agility, components	143
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A QUANTITATIVE MODEL FOR ASSESSING VISUAL SIMULATION SOFTWARE ARCHITECTURE

Robert W. Harder
Major, United States Air Force
B.S., Oregon State University, 1998
M.S., Air Force Institute of Technology, 2000

Submitted in partial fulfillment of the
requirements for the degree of

**DOCTOR OF PHILOSOPHY IN
MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)**

from the

**NAVAL POSTGRADUATE SCHOOL
September 2011**

Author: Robert W. Harder

Approved By:

Rudy Darken
Professor of Computer Science
Dissertation Supervisor

Ted Lewis
Professor of Computer Science

Richard Riehle
Professor of Practice
Software Engineering

Arnold Buss
Research Associate Professor
of MOVES

LtCol Jeff Boleng, PhD
Associate Professor of Computer Science
U.S. Air Force Academy

Approved By: Mathias Kölsch, Chair, MOVES Academic Committee

Approved By: Peter Denning, Chair, Department of Computer Science

Approved By: Douglas Moses, Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The U.S. military is the largest single user of simulation in the world, and our visual simulations can be software-intensive systems with a lifespan of many years. Managers of these simulations need tools to help them make better decisions at the architectural level. Currently, no such quantitative models with supporting metrics exist for this purpose. There are properties that are held as positive characteristics in visual simulation architectures. Visual simulation architectures can be distinguished from one another based on three characteristics: (1) openness, as defined by the use of standards, licensing, and support of innovation; (2) reuse, as defined by the potential of being used in subsequent projects; and (3) agility, as defined by the ease with which software can be integrated, reconfigured, or repurposed. In this research, we propose quantifiable models to measure openness, reuse, and agility, and claim that the models adequately distinguish visual simulation frameworks from one another. Furthermore, we claim that these models can enhance military acquisition decisions. The results show that application of the metrics offers a level of granularity that is useful in identifying key differences in simulation frameworks that could have profound downstream implications.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3D	Three Dimensional
AFI	Air Force Instruction
AMSMP	Acquisition Modeling and Simulation Master Plan
ANSI	American National Standards Institute
API	Application Programming Interface
ASI	Application Scripting Interface
C4ISR	Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance
CARDS	Central Archive for Reusable Defense Software
CBO	Coupling Between Object Classes
CK	Chidamber and Kemerer
COTS	Commercial Off-the-Shelf
DAC	Data Abstraction Coupling
DIT	Depth of Inheritance Tree
DSRS	Defense Software Reuse System
DoDD	Department of Defense Directive
DoD	Department of Defense
FOSS	Free and Open Source Software
GPL	GNU Public License
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ILS	Integrated Library Systems
IT	Information Technology
LCOM	Lack of Cohesion in Methods
LGPL	Lesser GNU Public License
MIT	Massachusetts Institute of Technology
MOSA	Modular Open Systems Approach
MPC	Message Passage Coupling

M&S	Modeling and Simulation
NATO	North Atlantic Treaty Organization
NOC	Number of Children
NPS	Naval Postgraduate School
ONR	Office of Naval Research
OO	Object Oriented
OSG	Open Scene Graph
OSJTF	Open Systems Joint Task Force
OSS	Open Source Software
OTDRP	Open Technology Development Roadmap Plan
PM	Program Manager
POSIX	Portable Operating System Interface for Unix
RAPID	Reusable Ada Product for Information System Development
REC	Response Entropy for a Class
RFC	Response for a Class
RNTDS	Restructured Naval Tactical Data System
SEA	Synthetic Environment for Assessment
SOA	Service Oriented Architectures
SOAP	Simple Object Access Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery, and Integration
U.S.	United States
VBS2	Virtual Battlespace 2
W3C	World Wide Web Consortium
WMC	Weighted Methods per Class
XML	Extensible Markup Language

ACKNOWLEDGEMENTS

Praise God from whom all blessings flow! Thank you to my wife, Gabrielle, and my seven kids, Emma, Riley, Max, Sam, Jane, Claire, and Kevin, who support me and bring me joy.

Thank you to my advisor, Dr. Rudy Darken, and my committee members, Dr. Ted Lewis, Dr. Richard Riehle, Dr. Arnie Buss, and Lt. Col. Jeff Boleng, PhD from the U.S. Air Force Academy, my pipeline sponsor.

Thank you to the late CAPT Gordon Nakagawa (USN), the coach of our NPS Marksmanship Team, and to Joachim Beer and the entire team for a healthy bit of distraction and improving my skills on the range.

Soli Deo Gloria!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This research is about choosing a visual simulation architecture in acquisition. "Architecture" is referring to such properties as object oriented, service oriented, component based, data driven, blackboard, and peer-to-peer. Acquisition is a big problem in the Department of Defense and is always being improved. In acquisition, simulation is sometimes used to help evaluate competing products. Visual simulations are a narrower field within that. Acquisition professionals have simulations to help them choose products, but they do not have quantitative tools to help them choose visual simulation architectures.

A. THESIS AND PROBLEM STATEMENTS

Visual simulation architectures can be distinguished one from another based on three objectives: (1) openness, as defined by code's use of standards, its licensing, and support of innovation; (2) reuse, as defined by code's ease of being used in subsequent projects; and (3) agility, as defined by the ease with which code can be integrated, reconfigured, or repurposed.

In our research, we found no existing quantitative models with metrics to help assess these objectives in visual simulation architectures. We developed three quantifiable models to measure openness, reuse, and agility and claim that the models adequately distinguish visual simulation frameworks from one another. Furthermore, we claim that these metrics are sufficient for improving military acquisition decisions.

B. BACKGROUND

The United States (U.S.) military is the largest single user of simulation in the world (McDowell, Darken, Sullivan, & Johnson, 2006). Military visual simulations are software-intensive systems that may have a long lifespan and address diverse military situations. In acquisition, a simulation sometimes supports a program, and sometimes the simulation

is the program. Military acquisition involves a wide range of products, services, tactics, doctrines, customers, vendors, timelines, and budgets that must be considered. Whether simulation is the central or supporting role, costs associated with simulation can be enormous. Therefore, having methods to help make good decisions are of critical importance.

Cost and safety concerns related to product development drive acquisition organizations to choose simulation, *e.g.*, synthetic environments, as a way to experiment with new ideas, mitigate risk, or objectively compare alternatives. When the human element is a critical part of the acquisition process, assessment becomes even more complex due to human capital availability, hardware and software changes, and the training needed to ensure success. Modern simulation tools and prototyping utilities are often expensive, inflexible, and proprietary.

In developing a new ship or airplane, a Program Manager (PM) may have the budget to create a massive simulation from scratch, but most in the acquisition community have to budget for simulation carefully. They have many programs to manage, many stakeholders to consider, and many pockets of experience scattered throughout the community and even within the program office. Their best hope is that they can build a simulation quickly and cheaply with good justification.

Even for well-funded acquisition programs, efficiency and cost effectiveness are important. Except for the case where the simulation itself is the program, every dollar spent on Modeling and Simulation (M&S) is a dollar not spent on other forms of engineering and product development. Defense M&S often has difficulty making a business case for itself because of this.

Program managers need simulation software that enables them to address a variety of needs, incorporate expertise from different sources, build trust in the simulations among users, answer questions quickly, and save money. It is rare for a program to start a major simulation project from scratch. Programmers use frameworks, Application Programming Interfaces (APIs), toolkits, libraries, etc., as a starting point for fast and cheap development. The understanding in the Department of Defense (DoD) about the importance of

architectures has improved, but there were no quantifiable tools to assist PMs in selecting a framework that represents a desirable architecture. The research presented in this dissertation did not reveal any previously-published scientific methodology or model using metrics for assessing simulation software. The requirement is legitimate and unfilled.

There are a variety of qualities in visual simulation software that a PM may desire. The three qualities of openness, reuse, and agility chosen for study in this dissertation are not comprehensive, but they are representative and supported in the literature. If code survives a long time, maintainability may also be important. Perhaps certain technical features are desired. Security is a concern—or should be—for many systems. Although a PM may desire all of these, some are more important to a project than others. Visual simulation fosters the sharing of data, whether models of equipment or terrain databases, so openness, reuse, and agility are of particular concern.

The specific needs for individual PMs and programs differ. Consider the following three scenarios with three PMs with different programs and different simulation needs (Table 1). One PM has a simulation for a major combat system. This is a long-term view. The system is expected to be around for many years, and participants will change over time. Another PM is in a different situation. Here there is a game based maintenance trainer for the F-22. If successful, the general is going to say, "That's great! Let's put it out for the Joint Strike Fighter too. Maybe we can get some of their funding." Finally, there is the PM with a UAV exercise coming up. There is a new camera to put on. Mission planners wonder how the flight profiles might change with a different sensor. It is no surprise that there is no one architecture or product that would be best for all of these situations, but what do the PMs have to help them choose an architecture? The three objectives of openness, reuse, and agility are good things on which to inform a decision.

The problem addressed in this dissertation is the absence of a quantifiable model, a tool, to aid PMs in selecting an appropriate visual simulation framework. The models developed here do not address all of the concerns a PM may have. Many cannot be addressed in a single dissertation. However, the models do provide a rigorous and quantifiable as-

Table 1: Characteristics of the three example programs drive different needs in the visual simulation software used in the programs.

Major Combat System	Game Based Maintenance Trainer	UAV Sensor Exercise
15 year timeline	Forks expected	Quick & dirty
Vendor longevity	Licensing	Focused question
Technology changes	Configuration	Pre-packaged
Maintenance costs	Manpower	

sessment of three important elements of visual simulation software: openness, reuse, and agility.

C. ORGANIZATION OF THIS DOCUMENT

Chapter II reviews the literature to show that there are no existing models available for assessing the important issues of openness, reuse, and agility for visual simulation. Regarding openness, the literature reveals a mixed set of issues. A taxonomy presented by Maxwell (2006) provides a good basis for our model. Openness is divided into three main attributes: standards, licensing, and innovation. A model presented by Anvaari and Jansen (2010) provides structure for our model. Regarding reuse, the literature reveals some established software metrics from S. Chidamber and Kemerer (1991). These metrics are respected and are connected to many positive software attributes including ease of reuse. Regarding agility, the literature reveals a less cohesive picture and no metrics on which to base a model.

Chapter III presents our openness model, how it was developed, with a study verifying the ability to distinguish between two visual simulation frameworks. Using the taxonomy of issues presented by Maxwell (2006) and the layers and operations presented by Anvaari and Jansen (2010), this research develops criteria for assessing openness on a three-axis system of layers, operations, and issues. The categorical ratings for each point can be weighted according to a user value system to provide numerical assessment, if needed. The

model is applied to two visual simulation frameworks representing two very different architectures. The model successfully differentiates the frameworks. It draws attention to important distinguishing attributes.

Chapter IV presents the reuse model, how it was developed, and a study verifying its ability to differentiate two visual simulation frameworks. Using the Chidamber and Kemerer (CK) metrics and empirical validation studies found in literature, this research develops criteria for the metrics. The categorical ratings for each metric can be weighted according to a user value system to provide numerical assessment, if needed. The model is applied to two visual simulation frameworks representing two very different architectures. The model successfully differentiates the frameworks. It is weaker than the openness and agility models.

Chapter V presents the agility model, how it was developed, and a study verifying its ability to differentiate twoscriptsize visual simulation frameworks. Unlike the reuse metrics, which can be calculated on static source code, agility metrics must be measured "in motion." There must be an act by a developer, and the effort must be estimated. This research develops metrics for estimating the effort of swapping out a portion of code and applies these metrics to two visual simulation frameworks representing two very different architectures. The agility model reveals a dramatic difference between the two architectures.

Chapter VI is the conclusion and presents a discussion about the research, a summary of findings, and considerations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LITERATURE REVIEW

This chapter presents our literature review and identifies openness, reuse, and agility as important features in software development for visual simulation. Common through all three attributes is a general agreement that they are desirable but that there exists a lack of detail about how to measure them.

Openness is defined by the use of standards, licensing, and support of innovation, is considered an important quality in visual simulation software. It is discussed in many papers inside and outside the visual simulation realm. Many people have opinions, assertions, and definitions for openness, but no models suitable for measuring the openness of visual simulation architectures could be found.

Reuse is defined by the ease of code being used in subsequent projects. Therefore, reuse is considered an important quality in visual simulation software. Effective code reuse has been a goal of computer programmers since subroutines were invented in 1949 for the EDSAC computer (Wilkes & Renwick, 1949). While we found many individual metrics for different aspects of reuse, we found no models of reuse that could encompass these metrics to supply a decision maker with differentiating results.

Agility is defined here as the ease with which code can be reconfigured, repurposed, or integrated. It is also an important goal in visual simulation software. As military threats change, software requirements also change (Lanman & Proctor, 2009; Scott, 2010), and therefore, simulation software must be easily repurposed for unanticipated use. Agility is the least rigorously defined and studied of the three features assessed in this dissertation. This literature was the most difficult for two reasons: (1) agility means different things to different people, and (2) our meaning of agility is sometimes only identifiable in an author's work by discerning the author's intent and examining his or her actual actions. No models suitable for measuring the agility of visual simulation architectures could be found.

A. OPENNESS

1. What is Openness?

There are many definitions and attributes of openness provided in the literature. A review of them reveals many common traits. While reviewing these many facets of openness, we use three overarching issues—standards, licensing, and innovation—that will support our openness model. This taxonomy of openness is used by Maxwell (2006) in his defense of the idea that value can be created through openness. All of the literature reviewed below focuses on one or more of these aspects of openness.

Gimenes, Silva, Reis, and Oliveira (2008) describe flight simulation environments for unmanned aerial vehicles. They address standards, licensing, and innovation with their definition of openness. Their definition includes defined APIs, communication protocols, data formats, viewing and altering source code, and community expansion through modules.

The Open Systems Joint Task Force (Open Systems Joint Task Force, 2004) addresses openness by providing guidance for program managers to implement a Modular Open Systems Approach (MOSA), which they call the "fundamental building block of joint integrated warfare systems." The Defense Acquisition Guidebook gives five MOSA principles, which address standards, licensing, and innovation: (1) establishing an enabling environment, (2) employ modular design, (3) design key interfaces, (4) use open standards, and (5) certify conformance (Defense Acquisition University, 2010).

a. Standards

Some literature highlights the role of standards with respect to openness. Here openness is defined by its relation to such issues as accessibility, integration, communication, and interoperability.

Describing the phases for developing a domain-oriented reuse library for the U.S. Air Force, Curfman (1993) addresses standards by identifying accessibility as an attribute of openness, which he connects to policy and communications. In the same

year Oswalt (1993), in his detailed report on "Current Applications, Trends, and Organizations in U.S. Military Simulation and Gaming," defines standardization as the "number of standards to which a particular simulation conforms" and recommends the military adopt commercial standards.

Writing for the Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR) Cooperative Research Program, Krygiel (1999) addresses standards by studying integration issues on large scale systems. She connects open systems with interfaces, data formats, hardware portability, and interoperability.

The Open Systems Joint Task Force (2004) says that standards must be "well defined, mature, widely used, and readily available" and that standards should be selected based on "maturity, market acceptance, and allowance for future technology insertion." Their preferences for the adoption of standards is (1) open standards, (2) de facto standards, and (3) government and proprietary standards.

Maxwell (2006) identifies standards with communication carrier and content. This he applies both to networking, with Transmission Control Protocol/Internet Protocol (TCP/IP) being an example of carrier and Hypertext Markup Language (HTML) being an example of content. He also calls attention to the ability of software to run on disparate hardware, as is the case with American National Standards Institute (ANSI) C or the Portable Operating System Interface for Unix (POSIX) standard.

b. Licensing

Open licensing refers to what a consumer is permitted to do with a piece of software. Licenses can be very restrictive, such as requiring a student-licensed copy of software not to be used for commercial purposes, or very open, such as the entire source for a system being made available for review and modification.

The struggle is between the creators, who have the right to control their work, and the users, who may see themselves as prevented from doing things with software that they might wish to do (Maxwell, 2006). The government's understanding of this bal-

ance is further evidenced in its opposition to certain kinds of monopolies but endorsing of limited-term, "mini-monopolies" with the patent and copyright systems.

The DoD Open Source Software (OSS) Frequently Asked Questions website (Department of Defense, 2011) identifies open source as "software for which the human-readable source code is available for use, study, re-use, modification, enhancement, and re-distribution by the users of that software."

Müller (2011) divides software licensing into seven categories: (1) Public domain, (2) Free Software, (3) Open Source Software, (4) Freeware, (5) Shareware, (6) Proprietary Software, and (7) Patented Software. Each of these has certain characteristics that our model should differentiate.

Different organizations differ on their distinction between *free* and *open source*. Richard Stallman, an advocate for free software, encourages people to think in terms of "free speech" rather than "free beer." His meaning of free software refers more to the rights granted to users of software (Stallman, 2010). The Open Source Initiative's definition of open source differs on some points from Stallman's definition of free (Open Source Initiative OSI, 2011). As a result, *free* is sometimes distinguished from *open source* (as with Müller's taxonomy above). Sometimes it is grouped together, as with the common acronym Free and Open Source Software (FOSS), which acknowledges that a distinction exists but that there is more in common than not.

The DoD is also careful to distinguish among different meanings of free and open. Chief Information Officer for the Department of Defense David M. Wennergren, in a memorandum dated 16 October 2009 with the subject "Clarifying Guidance Regarding Open Source Software (OSS)," clarifies that the DoD Instruction 8500.2, which limits the use of software with "limited or no warranty," does not apply to open source software, for which anyone could provide maintenance and a warranty because the source code is available. David Wheeler, who helped draft this memorandum, pointed out in an interview that freeware is no-cost, while closed-source is software for which there is no one to provide maintenance or a warranty. Therefore, open source software cannot be considered free-

ware (Schwartz & Phipps, 2011). The memorandum goes on to explain that "In almost all cases, OSS meets the definition of 'commercial computer software"' and provides six legal references. Because contractors are generally required to prefer "commercial computer software" over writing their own software from scratch, this clarification gives contractors greater latitude in selecting open source software in their work.

Maxwell (2006) recalls that in the early days of computing in the 1950s and 1960s, software was written primarily in academic settings where sharing ideas and information was expected but that it was a community norm, not a legal requirement. As people began to think in terms of owning software, open source licenses, which grant explicit permission to view and modify source code, arose. Open source licenses focus on the rights of the users and widest possible dissemination.

c. Innovation

Open innovation refers to a community being able to collaborate and share ideas. The moniker "innovation" is not always used in the literature. The innovation that people desire from openness is revealed in words like *participation*, *sharing*, and *collaboration*.

In an article entitled "The Many Faces of 'Open,"' Updegrove (2005) relates attributes of openness that he claims are "generally conceded." Three features of openness to which he calls attention are participation (who is involved), process (participants ability to influence the outcome), and terms (intellectual property rights). In this Updegrove acknowledges both the innovation and the licensing aspects of openness.

For Maxwell (2006), the defining characteristics of open innovation are collaboration and sharing. He warns that this should not be confused with a lack of intellectual property or the abolishment of compensation but shows that new forms of compensation emerge when open innovation takes hold. Open innovation is a community being able to collaborate and share ideas and software.

2. Why Openness is Important

Openness is important to software development and visual simulation software in particular. A review of the literature reveals many advantages of openness. A common thread is that big ideas from individuals or small teams can find their way to a wide audience when openness is encouraged. Innovators are able to leverage the work done by others, standing on the shoulders of giants.

a. Benefits

Openness is of immense importance in the world of the acquisition professionals who seek to find the best solutions for the government. They may need to merge solutions from many sources and may find themselves having to integrate partial solutions to achieve their goals. In the case of simulation, with the right software framework, they may be able to forge their own simulations, leveraging the work of others and contributing back the portions unique to their domain.

Describing the process of modifying Commercial Off-the-Shelf (COTS) games for military use, Fong (2004) notes the shift from paying game developers huge sums of money and submitting to stringent licensing agreements to modifying existing games with *mods*. He cites many benefits that this example of open innovation brings. For the game developer, third party mods extend the shelf life of their games bringing economic benefit. Those writing mods benefit from a low barrier to entry. With low risk and low cost, they get to leverage sophisticated game technology. Mods offer quicker turnaround time in bringing new capabilities to bear than creating a new game from scratch. Experienced soldiers, marines, sailors, and airmen in the community of gamers can identify inaccuracies and report them resulting in higher fidelity models. Fong identifies the biggest obstruction is a lack of source code and hopes that more developers will embrace the open source movement: "If more game developers adopt the same mantra of free information access, it will pave the way for more extensive modifications of COTS games to meet specific military interests."

The Open Systems Joint Task Force (2004) lists many benefits of openness for program managers. They encourage program mangers to embrace MOSA because it is an enabler to help them make affordable changes, employ spiral development, develop integrated roadmaps for their projects, adapt to evolving threats, leverage commercial investment, reduce development time, improve interoperability, mitigate risk from obsolescence, and mitigate risk of a single supply source, to name a few.

Wichmann (2002) reports on the state of open source software in businesses and points out that openness in the standards creation process is critical to reduce the possibility that a standard is used to keep certain players out of a market. Open standards increases competition and helps ensure a level playing field for all participants.

Open standards encourage participation by many parties and reduce the likelihood that only a single party's interests are represented. One company denying access to a specification to a competitor may be attractive to the company in power, but it is a disadvantage to consumers who may resist being locked into one provider because of a lack of open standards. Greater participation and greater adoption spurs greater innovation of benefit to consumers (Maxwell, 2006).

The Economist (2005) noted that open standards "allow and promote unexpected forms of innovation." They cite several examples where people have made *mash-ups* that create value by joining information from one website with information from another. This also serves to boost the popularity of the source websites. The economic benefit provided by mash-ups is necessary because, "if the information being mashed is useful, it is probably expensive for the originated sites to put on the web in the first place."

WaughPartners and OSSWatch (2007) show that openness impacts sustainability, applicability, interoperability, and trustworthiness. Sustainability is improved for data by open data standards and in software by open source licensing. This helps protect against data loss and software obsolescence. Applicability (who is able to benefit from software) is improved by replacing a single point of control—and their single set of requirements—with open access and collaboration allowing more people to add value to

the system. Interoperability is improved with open communication standards. TCP/IP is a prime example. Trustworthiness (knowing that a system will perform as documented) is improved by open source licensing where interested parties can perform their own audits on software.

The Open Technology Development Roadmap Plan (OTDRP), prepared for the Deputy Under Secretary of Defense, identifies open standards and open interfaces as key elements in the success of DoD software (Herz, Lucas, & Scott, 2006). It states that the "DoD must pursue an active strategy of open interfaces, modularity, and reuse" and outlines a strategy to combine "salient advances" in the following areas, which are directly addressed by this research (Figure 1).

Open Technology Development combines salient advances in the following areas:

1. ***Open Standards and Interfaces***
2. ***Open Source Software and Designs***
3. ***Collaborative/Distributive culture and the and online support tools***
4. ***Technological Agility***

Figure 1: The advances recommended by the Open Technology Development Roadmap plan (Herz et al., 2006) are supported directly by this research.

The OTDRP also identifies advantages that openness (and reuse and agility) in software development contribute to national security:

- Enhances agility of Information Technology (IT) industries to more rapidly adapt and change to user needed capabilities.
- Strengthens the industrial base by not protecting industry from competition. Makes industry more likely to compete on ideas and execution versus product lock-in.
- Adoption recognizes a change in our position with regard to balance of trade of IT.

- Enables DoD to secure the infrastructure and increase security by understanding what is actually in the source code of software installed in DoD networks.
- Rapidly respond to adversary actions as well as rapid changes in the technology industrial base.

Several authors write that openness fosters interoperability, spurs competition that benefits consumers, increases participation, increases opportunities for success, and avoids vendor lock in (Davis & Anderson, 2004; Maxwell, 2006; McDowell et al., 2006; Scott, 2010). Maxwell (2006) acknowledges that some people argue that open standards may reduce efficiencies that tightly integrated proprietary systems can offer and that open standards may reduce innovation because developers are no longer forced to think outside the box as they are forced to work around proprietary technology. This is a poor argument, and he asserts that standards permit innovation to be focused where the real value lies, in integrating systems.

Regarding innovation, *Democratizing Innovation* (Hippel, 2005) and *The Wealth of Networks: How Social Production Transforms Markets and Freedom* (Benkler, 2006) explore the value added to individuals and a society by collaboration and sharing. They show that win-win scenarios are possible when innovative ideas are shared. In this way innovation mirrors money in an old proverb that might be updated to read, "Innovation is like manure. Unless you spread it around, it doesn't do much good."

Open innovation also blurs the line between producer and consumer. Consumers know their needs but cannot always articulate them well. However, give someone capable tools, and they may solve their own problem. The online auction site eBay is an example of a company that provides the means for users to solve their own problems—and thousands of people now make their living with their own "eBay Store."

b. Mandates

Not to be overlooked are government mandates related to openness. Be assured that these mandates are in place because of the benefits provided by openness, but as mandates or government "suggestions" they receive special recognition here.

The Clinger-Cohen Act of 1996 directed the U. S. Government to operate more business-like with respect to information systems. (Clinger & Cohen, 1996) Accordingly Department of Defense Directive (DoDD) 5000.01 requires the use of modular, open systems. Enclosure 1.1.27 states that, "Acquisition programs shall be managed through the application of a systems engineering approach that optimizes total system performance and minimizes total ownership costs. A modular, open-systems approach *shall be employed, where feasible*" (emphasis added) (Department of Defense, 2003). DoD Instruction 5000.02 follows up in Enclosure 12.8: "MODULAR OPEN SYSTEMS APPROACH (MOSA). Program managers shall employ MOSA to design for affordable change, enable evolutionary acquisition, and rapidly field affordable systems that are interoperable in the joint battle space" (Department of Defense, 2008).

Individual services have followed up with their own guidance directing the use of modular, open systems. Expanding on DoDD 5000.01, then United States Assistant Secretary of the Navy John J. Young, Jr., in a letter dated 5 August 2004 with the subject "Naval Open Architecture Scope and Responsibilities," wrote that he was initiating "an effort to establish open architecture principles as the basis for all war fighting systems development and maintenance" (Young Jr., 2004). The Deputy Chief of Naval Operations Rear Admiral M.J. Edwards, in a letter dated 23 December 2005 with the subject "Requirement for Open Architecture (OA) Implementation," established "the requirement to implement Open Architecture (OA) principles across the Navy Enterprise" and cited as one justification the need to "implement agile changes that support rapidly evolving requirements" (Edwards, 2005).

The "Clarifying Guidance Regarding Open Source Software (OSS)" memorandum referred to earlier identifies open source software as an enabler to "anticipate new

threats and respond continuously to changing environments" (Wennergren, 2009). This memorandum identified many advantages of open source software such as increased security through peer review, faster response times through unrestricted ability to modify source code, reduced barriers to entry and exit for vendor participation, and cost savings by reducing per-seat licensing and a shared maintenance burden. This memorandum clarifies many misunderstandings in the DoD about open source software.

DoD Instruction 5000.2-R *Mandatory Procedures for Major Acquisition Defense Programs and Major Automated Information Systems Acquisition Programs* identifies openness in system design as crucial to enable full and open competition (Office of the Under Secretary of Defense, 2002). Regarding standards, it states that "M&S standards facilitate reuse... and reduce cost by providing approved solutions to common problems."

The *OSD Acquisition Modeling and Simulation Master Plan (AMSMP)* directs the development of open standards in systems engineering and architecture modeling as well as standards for distributed, simulated environments (Office of the Under Secretary of Defense, 2006).

In *Modeling and Simulation in Manufacturing and Defense Systems Acquisition* the National Research Council recommends a culture of collaboration, part of openness, in DoD acquisition (National Research Council, 2005).

3. How Openness is Measured

There has been some effort to quantify openness at different levels. The literature has both one-off suggestions for openness metrics as well as in-depth surveys to assess licensing details. This literature review did not reveal any openness models that would meet our requirements for differentiating visual simulation architectures.

Regarding standards, the Open Systems Joint Task Force (2004) encourages program managers to use specific program measurements to gauge a program's progress in implementing MOSA. It even goes so far as to suggest a metric for openness: "For example, the percentage of key interfaces defined by open standards could be used as a metric to measure the degree of system openness."

WaughPartners and OSSWatch (2007) present their "first draft" of an openness evaluation model and provide a survey with 46 questions regarding licensing, standards, knowledge, governance, and marketplace along with numeric values for the responses to the questions in order to calculate an openness value for each of the five categories. They apply their model to three open source Unix-like kernels and three databases (two open source and one proprietary). The issues addressed in this work fit roughly within our taxonomy of openness—standards, licensing, and innovation—but while it addresses openness in great detail, it addresses software projects as a whole and does not distinguish between the different parts of a system and degrees of openness within each part, which we show in this dissertation is a strong differentiator in visual simulation.

Anvaari and Jansen (2010) evaluate five mobile phone operating systems and introduce evaluating openness at a finer grained level. They evaluate the software along two axes, which they call layers and factors. Layers refers to the function and scope of the software pieces. Factors refers to possible development operations performed on the software. Their evaluation consists of determining whether each factor is possible to perform at each layer and the associated licensing implications. Standards and innovation are not part of their evaluation.

Focusing on licensing, Müller (2011) evaluates 20 Integrated Library Systems (ILS) for the purpose of finding the best open source programs for library use. He throws out all systems that he does not qualify as open source. To determine what is open, he measures the "correlation between the practices within the community and the terms associated with free or open software license" and divides software licenses into seven categories from public domain to patented. Differentiation by license is his first concern. He continues his ILS evaluation by considering the communities behind each project and almost 800 specific functions and features related to library use.

4. Summary of Openness

Openness has many facets, but the three major headings of standards, licensing, and innovation capture most people's concerns and represent the openness we wish to measure

in visual simulation architectures. In all the literature reviewed, no models suitable for measuring the openness of visual simulation architectures could be found, though some features on which a model could be built are present.

B. REUSE

1. What is Reuse

There are varying forms of reuse and meanings used in the literature, and since some confusion may arise when definitions are too narrow, we present some explanation of the term *reuse*. The simplest forms of reuse have been around the longest and have the most widespread use. More sophisticated forms of reuse arose later and sometimes have limited applicability or are tied to specific architectures. Ultimately, the literature focuses on reuse as a measure of code quality. Many metrics have been presented to this end, and most assume Object Oriented (OO) programming.

Ambler (1998) breaks out reuse into eight different types: (1) Code, (2) Inheritance, (3) Template, (4) Component, (5) Framework, (6) Artifact, (7) Pattern, and (8) Domain Component reuse. Making use of the visual simulation frameworks discussed in this dissertation would involve *code*, *inheritance*, and *framework* reuse.

Reuse through patterns is considered a high form of reuse because it refers not to actually reusing code but reusing ideas or approaches to solving problems. In this sense patterns permit reuse to happen even across programming languages. In the landmark book *Design Patterns: Elements of Reusable Object-Oriented Software* the so-called "gang of four" identify such common patterns as Singleton, Adapter, Iterator, and Observer (Gamma, Helm, Johnson, & Vlissides, 1995).

J. Lewis (2006) identifies several examples of coarse reuse in military simulations. These include purchasing off the shelf games, *e.g.*, *Microsoft Flight Simulator* and *Falcon 4.0*, modifying existing games, *e.g.*, *Spearhead II* and *Marine Doom*, and paying to have game companies develop custom simulations with their underlying engines, *e.g.*, *America's Army*.

McIlroy (1968), at a now-famous North Atlantic Treaty Organization (NATO) conference in Garmisch, Germany, proposed a market for "mass produced software components" where pieces of code could be bought and sold. He recognized that no one company or even industry would be able to produce a full complement of quality software but that each could contribute its best work. Although McIlroy's specific vision has not materialized, parts of his ideas can be seen in the emergence of both a multi-billion dollar commercial software industry as well as the open source software community.

Jansen, Brinkkemper, Hunink, and Demir (2008) define two kinds of reuse that describe the developer more than the code. Pragmatic reuse is the extension of third party software that was not found with any formal procurement method and may not have been designed with reuse in mind. Opportunistic reuse is extending software *with* third party software that was not meant to be integrated or reused. Though others prefer more deliberate and systematic reuse (Morad & Kuflik, 2005; Ommering, 2005), Jansen et al. show how two companies benefit from even this kind of *ad hoc* reuse.

2. Why Reuse is Important

Reuse is considered by many to be a key to improving software productivity and quality (Biggerstaff & Richter, 1987; Kim & Stohr, 1992; Mili, Mili, & Mili, 1995), and reuse is mandated or strongly encouraged in the DoD. However, after many years of research and advances in techniques and metrics, reuse is still not as common as many people would like (Biggerstaff & Richter, 1987; Herz et al., 2006; Garlan, Allen, & Ockerbloom, 2009; Scott, 2010).

a. Benefits

For most people it is intuitive that code reuse is a good thing, and when pressed, two main reasons are likely to emerge: time and money (Washizaki, Yamamoto, & Fukazawa, 2003; Haefliger, Krogh, & Spaeth, 2008). Of course, the two are related, and a saving of either one is of great interest (Becker, 1965). Development savings can range

from 50–100% with typical savings being about 80%, which can pay off even when it costs extra to write the initial code with reusability in mind (Poulin, 2006).

Reuse can reduce the effort required of developers (Mili et al., 1995; Davis & Anderson, 2004; Ragab & Ammar, 2010) by amplifying the software developer's capabilities and reducing the number of symbols in a system (Biggerstaff & Richter, 1987) resulting in higher quality software (Mili et al., 1995). Brutzman, Zyda, Pullen, and Morse (2002) write, "Interoperable reuse is essential for feasibility, life-cycle supportability, fundability, and product flexibility."

Analyzing his company Philips, which manufacturers software-intensive hardware such as medical systems and consumer electronics, Ommering (2005) cites three main challenges, which he shows are improved by good reuse: (1) increasing complexity in the software, (2) growing diversity in products, and (3) a decrease in allowed development time. These challenges should be familiar to any program manager in the Department of Defense. He finds that their systematic and component based approach to reuse results in a more manageable software base than the "opportunistic" reuse that marked the early days of electronics manufacturing. The component subsystems have a longer lifespan than the actual products they support, and Philips has developed a set of golden and silver rules with varying degrees of effectiveness at reducing unintended consequences. Ommering and others (Biggerstaff & Richter, 1987) observe the balance in writing software general enough to be reused but specific enough to be useful. He believes that their deliberate efforts at systematic reuse have significantly benefitted the company.

Software is often not reused in the DoD (Herz et al., 2006; Scott, 2010). This results in wasted funding with multiple development efforts. This also results in slower development times making it harder for the DoD to respond to new missions and emerging threats.

b. Mandates

The government has opinions regarding reuse as well, and military services have instituted programs to promote and manage code reuse. Early efforts include the Air

Force's Central Archive for Reusable Defense Software (CARDS), the Navy's Restructured Naval Tactical Data System (RNTDS), the Army's Reusable Ada Product for Information System Development (RAPID) program, and the Defense Software Reuse System (DSRS) (Therriault & Van Nederveen, 1994).

Air Force Instruction (AFI) 33-114 states, "Software reuse benefits the Air Force through increased developer productivity, improved quality and reliability of software-intensive systems, enhanced system interoperability, lowered program technical risk, and shortened software development and maintenance time" (U.S. Air Force, 2004).

In "Open Technology Development (ODT): Lessons Learned and Best Practices for Military Software" Scott (2011) emphasizes the need for code reuse as a necessary means for developing software quickly and inexpensively. The OTDRP also recommends reuse. The AMSMP assumes the need for reuse and lays out requirements for systems enabling the discovery of reusable software and data (Office of the Under Secretary of Defense, 2006).

3. How Reuse is Measured

We can divide the measurement of reuse into two categories: actual reuse and potential for reuse. Actual reuse refers to completed projects that have reused code from previous projects, such as the OTDRP suggesting DoD program managers count the number of times a software component has been used by multiple (acquisition) programs (Herz et al., 2006). Potential for reuse, with which this dissertation is more concerned, refers to the ease with which code might be reused, based on various qualities of that code.

Briand, Daly, and Wust (1999) defined two kinds of attributes that help describe software: internal and external. Internal attributes can be defined in terms of the software itself (*e.g.*, lines of code). External attributes cannot be measured solely in terms of the software (*e.g.*, comprehensibility). Some external factors that affect reusability include comprehensibility and maintainability (Cho, Kim, & Kim, 2007).

Complexity, coupling, and cohesion are three internal attributes that support reusability and are related to comprehensibility, maintainability, quality, and the separation of function and purpose (Table 2).

Selecting among OO metrics is a weighty task. Xenos, Stavrinoudis, Zikouli, and Christodoulakis (2000) list over 89 metrics that can and have been applied to OO software and more metrics have been published since then (Table 3). Despite the many metrics put forth, only a few have lasted beyond a few papers.

S. Chidamber and Kemerer (1991) proposed metrics to address object oriented software specifically. Their work was immediately taken up by others. Li and Henry (1993a) refine some ambiguities in the CK metrics and propose two more detailed metrics to replace the CK metric regarding class coupling.

Again regarding coupling, Martin (1994) distinguishes between individual classes and categories of classes and introduces Instability, a ratio describing how many classes inside a category depend on classes outside the category.

The stability of the software involved in the coupling also affects the degree to which coupling may have a practical effect (White, 1994; Hitz & Montazeri, 1995). Coupling to basic language implementations such as integers are less troublesome than coupling to small foundation classes such as string or date, and all of these couplings are preferred over coupling to classes in the problem domain.

The most enduring software metrics related to reusability and other qualities come from S. R. Chidamber and Kemerer (1991, 1994). Their six "CK" metrics have been often put to the test and are still used today (Cho et al., 2007). They address concerns that too many software metrics, especially for object oriented design, do not have solid theoretical basis (Kearney, Sedlmeyer, Thompson, Gray, & Adler, 1986), lack theoretical rigor (Vessey & Weber, 1984), refer to procedures rather than objects (Henry & Kafura, 1984), and do not possess appropriate mathematical properties for producing "normal predictable behavior" (Prather, 1984; Weyuker, 1988). The CK metrics are Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object

Table 2: Researches agree that complexity, coupling, and cohesion affect code comprehensibility, maintainability, quality, and therefore reusability.

Paper	Quote
Mills, 1988	The results indicated that module coupling was an important factor in determining the quality of the resulting product.
Wand & Weber, 1990	It is generally believed that system decompositions which have "loosely-coupled" subsystems are easier to understand than system decompositions which have "tightly-coupled" subsystems.
Devanbu, Brachman, & Selfridge, 1991	Thus, this lack of knowledge among developers leads to a vicious cycle where the system becomes progressively more complex, and thus harder to know.
S. Chidamber & Kemerer, 1991	Excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse.
Sharble & Cohen, 1993	The recognized achievement of OOSD is the production of software that is less complex, and is therefore easier to maintain and extend, and can be more easily reused.
Sharble & Cohen, 1993	Excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse.
Hitz & Montazeri, 1995	Software engineering experts assure that designs with low coupling and high cohesion lead to products that are both, more reliable and more maintainable.
Briand, Morasca, & Basili, 1996	Lower complexity is believed to provide advantages such as lower maintenance time and cost.
Briand, Daly, Porter, & Wust, 1998	Some measures, in particular coupling and inheritance ones, are shown to be significantly related to the probability of detecting a fault in a class during testing.
Allen & Khoshgoftaar, 1999	When used in conjunction with measures of other attributes, coupling and cohesion can contribute to an assessment or prediction of software quality.
Tang, Kao, & Chen, 1999	Excessive coupling indicates weakness of module encapsulation and may inhibit reuse.
Agrawal, Bayardo, Gruhl, & Papadimitriou, 2002	Such loose-coupling of distributed components reduces coordination overhead, fostering faster parallel development.
Open Systems Joint Task Force, 2004	Decoupling modules eases development risks and makes future modifications easier.
Xu, Qian, & He, 2006	The decoupling metrics can be used to measure and evaluate the decoupling attributes of a distributed, service- oriented software architecture that has very significant impacts on the understandability, maintainability, reliability, testability, and reusability of software components.
Cho et al., 2007	Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application.
Offutt, Abdurazik, & Schach, 2008	Software coupling can be used to estimate a number of quality factors, including maintainability, complexity, and reliability.

Table 3: A sampling of the over eighty metrics which have been proposed to measure software (Xenos, 2000).

Metric	Description
AHF	*Attribute Hiding Factor* is the ratio of the sum of inherited attributes in all system classes under consideration to the total number of available classes attributes.
CEC	*Class Entropy Complexity* measures the complexity of classes based on their information content
CLM	*Comment Lines per Method* measures the percentage of comments in methods.
DAM	*Data Access Metric* is the ratio of the number of private attributes to the total number of attributes declared in the class.
FOC	*Function Oriented Code* measures the percentage of non object–oriented code that is used in a program.
INP	*Internal Privacy* refers to the use of accessory functions even within a class.
MAA	*Measure of Attribute Abstraction* is the ratio of the number of attributes inherited by a class to the total number of attributes in the class.
MHF	*Method Hiding Factor* is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.
NAD	*Number of Abstract Data types* is the number of user-defined objects used as attributes in a class that are necessary to instantiate an object instance of the class.
NCM	*Number of Class Methods in a class* measures the measures available in a class but not in its instances.
NPM	*Number of Parameters per Method* is the average number of parameters per method in a class.
PCM	*Percentage of Commented Methods* is the percentage of commented methods.

Classes (CBO), Response for a Class (RFC), Lack of Cohesion in Methods (LCOM). These metrics will be examined in greater detail in the Chapter IV: Reuse.

Rosenberg and Hyatt (1995) summarize several papers and books that propose metrics to measure various software qualities including "understandability, reusability, and maintainability," all three of which are linked in their estimation. They identify supporting papers that confirm the relationship between the metrics and the various software qualities. Of the metrics that pertain to understandability, reusability, and maintainability, they cite Size with four supporting works, Comment Percentage with one supporting work, WMC with four supporting works, RFC with four supporting works, LCOM with five supporting works, CBO with six supporting works, DIT with five supporting works, and NOC with four supporting works.

Kitchenham (2010) surveys the literature again and finds that CK metrics dominate the research. Although she and a few others take issue with some of them, these metrics are the most studied and most understood metrics that we can "reuse" to help us measure reuse.

The CK metrics have been validated by a number of studies (Li & Henry, 1993b; Basili, Briand, & Melo, 1996; S. Chidamber, Darcy, & Kemerer, 1998; Tang et al., 1999; Cartwright & Shepperd, 2000; Olague, Etzkorn, Gholston, & Quattlebaum, 2007). Most of these studies correlated the CK metrics to the fault-proneness of the code, a response variable that could be measured by examining the change history of a codebase. These validation studies also provide an empirical basis for estimating expected values of the metrics in software.

4. Summary of Reuse

Reuse has been estimated by many metrics by many people, but the most enduring are the CK metrics. The literature overwhelmingly speaks of metrics for reuse in terms of code quality. This is not the meaning of reuse that we originally intended, and this forced the introduction of agility as a third model. In all the literature reviewed, no models

suitable for measuring the reuse of visual simulation architectures could be found, though some metrics on which a model could be built are present.

C. AGILITY

1. What is Agility

We refer to agility specifically as software being easily reconfigured, repurposed, or integrated, but there are many other definitions to sift through in the literature. Qumer and Henderson-Sellers (2006b) writes, "there is no rigorous or complete definition of agility." Dove (1994) writes, "agility is a very seductive word" and describes a litany of "personal definitions" that may accompany it: cycle time reduction, customization, streamlining, reengineering, learning organization, productivity. Scott (2010) uses the term *adaptability*. TechWeb (2008) uses the expression, "react quickly to changing market dynamics." Several authors, especially in the *Journal of Defense Modeling and Simulation*, instead use the term "composability" to refer to agility (Yilmaz, 2004; Davis & Anderson, 2004). The Defense Acquisition Guidebook links agility to integration and optimization (Defense Acquisition University, 2010).

For the Office of Naval Research (ONR), Tangney (2009) desires to make "interesting perturbations" of Naval tasks under which they would like to make measurements in a calibrated Synthetic Environment for Assessment (SEA). This reconfiguring of an SEA represents agility.

Agile Methods is different from agility. Agile Methods is a movement in software development that favors collaboration, interaction, and responsiveness over "documentation driven, heavyweight software development processes" (Beck, Cockburn, Jeffries, & Highsmith, 2001; Cohen, Lindvall, & Costa, 2004). Agile Methods may also have value in visual simulation development, but that is not the focus of this dissertation.

2. Why Agility is Important

Scott (2010) defends the accusations that the U.S. government lacks imagination but instead contends that "we are simply unable to deploy new ideas as effectively or as quickly

as we could." He cites industrial examples of large-scale agility while the DoD spends tens of billions of dollars annually that is "rarely reused and difficult to adapt to new threats." Lynn III (2010) writes, "Cyberwarfare is like maneuver warfare, in that speed and agility matter most."

a. Benefits

Davis and Anderson (2004) give several reasons why agility in software is important to defense M&S. They provide their reasons as assertions, acknowledging that a body of literature and the common sense of practitioners agree. Software modules that are easily reconfigured, repurposed, or integrated are also easier to build at the *creation* phase. Such software tends to be easier to *understand*. It makes *testing* easier when modules can be pulled out and tested on their own. *Cost* can be reduced.

McDowell et al. (2006) point out that agility also helps mitigate risk when technologies mature at different rates. They also highlight the need to adapt simulations to different genres of simulations and different real world domains from air to land to sea.

In studying Service Oriented Architectures (SOA) in businesses, several authors highlight the positive impact of agility (TechWeb, 2008; Feig, 2008). Agility is linked to efficiency in software development and faster time to market.

The OTDRP (Herz et al., 2006) highlights the need to adapt to new "trends, capabilities, and practices." By falling behind in software, the DoD has seen costs spiraling up and a loss of useful software to those on the ground. It encourages the DoD to create market incentives to increase agility, but they offer no metrics with which to achieve this.

b. Mandates

As mentioned in Section b, the Deputy Chief of Naval Operations saw a need to "implement agile changes that support rapidly evolving requirements" and to that end wrote a policy letter directing open architecture principles across the Navy (Edwards, 2005).

The AMSMP links a loss of agility in software to a lack of ability to develop live-virtual-constructive environments that exploit the full range of hardware, software, ranges, equipment, and other resources that are available. The result is simulations that are less capable than they could be (Office of the Under Secretary of Defense, 2006).

3. How Agility is Measured

We found very little literature regarding any measurements of agility. The metrics that we did find are tangentially related and help provide context, but our literature review did not reveal any agility models that would meet our requirements for differentiating visual simulation architectures.

Qumer and Henderson-Sellers (2006b) measures agility in organizations using Agile Methods. He develops four dimensions to measure. One dimension that is closest to our needs is *agility characterization* with metrics for Flexibility, Speed, Leanness, Learning, and Responsiveness. These metrics have a value of 0 or 1 for various phases and segments depending on the answer to a yes/no question provided for each metric. For example, the question associated with Flexibility is, "Does the method accommodate expected or unexpected changes?" During a planning phase, flexibility may be a 1 but leanness may still be 0. Adding up these values, and dividing by the total number of measurements taken, gives a fractional value indicating degree of agility. Qumer and Henderson-Sellers (2006a) use this methodology to analyze two Agile Methods known as XP and Scrum for the purpose of helping organizations select among competing Agile Method approaches.

4. Summary of Agility

Agility means different things to different people. Therefore, a review of the literature required casting a much broader net over related topics. Still in all the literature reviewed, no models suitable for measuring the agility of visual simulation architectures could be found.

D. SUMMARY OF LITERATURE REVIEW

This chapter reviewed literature that identifies openness, reuse, and agility—important features in software development, especially as it relates to visual simulation. There is some agreement in definitions and importance, and there were related efforts at measuring these factors, but we found no quantitative models suitable for our use in measuring openness, reuse, and agility in visual simulation architectures.

What the literature did reveal are some foundations on which we can build assessment models with confidence. These features of openness, reuse, and agility are not new, and there is great interest in them. These foundations, built over many years by many people, enable assessment models to be built for this dissertation. The literature provided least benefit with respect to assessing the agility of visual simulation frameworks.

III. OPENNESS

In this chapter, we develop and apply a model for assessing openness based on the definition and need established in Chapter II: Literature Review. The model borrows from taxonomies and methodologies presented in literature and presents a new composite model that is subsequently shown to differentiate between two visual simulation frameworks.

A. DEVELOPING THE OPENNESS MODEL

To assess visual simulation frameworks, we want a model that can differentiate between the parts within a framework, the actions we might take, and do all this across various issues associated with openness. We assess on a three axis system with four layers of software, three operations of development, and three issues in openness.

1. Layers and Operations

Anvaari and Jansen (2010) develop a model for assessing the openness of mobile phone operating systems. For each of four layers of the architecture they consider three factors, whether or not the factors are possible, and the licensing restrictions associated with them.

Their breakdown of layers and factors can be applied to visual simulation software despite the fact that it was developed for mobile phone software. The four layers they identify are Kernel, Middleware, Native Applications, and Extended Applications. The three factors they identify are integrating (use existing components), extending (enhance functionality of components), and modifying (replace a component) the platform (Figure 2).

We use the four layers given by Anvaari and Jansen (2010) and define the layers in the context of visual simulation frameworks:

- **External Applications.** The final simulations and applications built with the framework.

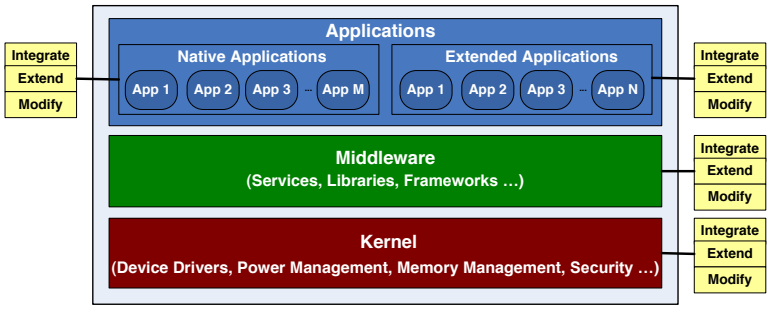


Figure 2: A model that considers the effort and permissions required to integrate, extend, or modify a platform (From Anvaari & Jansen, 2010).

- **Internal Applications.** Applications or tools included with the standard distribution used either in the building or running of the simulation.
- **Middleware.** The standard pieces that are included with the framework, often called "foundation classes" in the case of a programming language.
- **Kernel.** The core engine that manages the simulation pieces.

The model does not include a holistic analysis of openness, but it measures whether or not an operation can be performed at a given layer and if so, the licensing restrictions therein. We retain the operational definitions of Integrate, Extend, and Modify:

- **Integrate.** To use the existing pieces of a layer via API, service call, source code inclusion, shared data object, and other software calling mechanisms. An example would be using the cURL libraries to make HTTP requests.
- **Extend.** To enhance the functionality of the pieces of a layer beyond just making use of that layer. An example would be to change the behavior of a camera-following-an-object routine.
- **Modify.** To replace or change the pieces of a layer. An example would be replacing the physics engine.

Anvaari and Jansen finish their model with a summary of the "possibility" measurements they made (Figure 3). Their summary provides both a quick look assessment for differences among the platforms as well as details that can be inspected at particular points along the axes.

	Android		Symbian		Windows Mobile		Blackberry		iPhone	
Factor	P	L	P	L	P	L	P	L	P	L
Integrate extended applications	Pc	Pn	Pc	Ps	Pc	Ps	Pc	Ps	Pc	Pa
Extend extended applications	Pc	Pn	Pc	Ps	Pc	Ps	Pc	Ps	Pc	Pa
Modify extended applications	Pc	Pn	Pc	Ps	Pc	Ps	Pc	Ps	Pc	Pa
Integrate native applications	Po	Pn	Pc	Ps	Np		Pc	Ps	Pc	Pa
Extend native applications	Po	Pn	Po	Ps	Np		Np		Np	
Modify native applications	Po	Ps	Po	Ps	Np		Np		Np	
Integrate middleware	Po	Pn	Po	Ps	Po	Ps	Po	Ps	Po	Pa
Extend middleware	Po	Pn	Po	Ps	Pc	Ps	Np		Po	Pa
Modify middleware	Po	Ps	Po	Ps	Np		Np		Np	
Integrate kernel	Po	Pn	Po	Ps	Po	Ps	Np			Pa
Extend kernel	Pc	Pn	Po	Ps	Pc	Ps	Np		Pc	Pa
Modify kernel	Pc	Ps	Po	Ps	Pc	Ps	Np		Np	

P = Possibility, L = Licensing Status, Po = Possible, Pc = Possible for some components, Np = Not possible, Pn = Permission is not needed, Ps = in some cases permission is needed, Pa = Permission is always needed

Figure 3: Anvaari & Jansen (2010) analyze mobile smart phone architectures with their openness model, which we use as a starting point for our model which is more all-encompassing of the notion of openness.

They find that not all organizations are concerned about all the layers and operations. For many developers, as long as they can integrate, extend, or modify their own extended applications, that is sufficient for them. This helps to explain why developers

continue to create iPhone (now iOS) apps despite the cries of open source advocates that the developers' hands are shackled on that platform. One developer humorously explains that "if a platform is open enough to make a lot of money for him, then the platform is interesting for him."

2. Issues

Maxwell (2006) defends the idea that value can be created through openness and presents a useful taxonomy of openness: open standards, open licensing, and open innovation. These three issues within openness form the basis for our third axis for analysis.

Open standards refer to the communication mode and content of a system. This might include a program's application programming interface (API), the data formatting and syntax offered in the Extensible 3D (X3D) graphics format, or the binary byte ordering of the Internet Protocol (IP). There is such a need for standards that there are many national and international standards bodies addressing various technical fields such as the Internet Engineering Task Force (IETF) for the Internet and the International Telecommunication Union (ITU) for telephone communication.

Open licensing refers to what a user is permitted to do with a piece of software. Licenses can be very restrictive, such as requiring a student-licensed copy of software not be used for commercial purposes, or very open, such as the entire source for a system being made available for review and modification.

Open innovation refers to a community being able to collaborate and share ideas and software. Big ideas from small parties can find their way to a wide audience when innovation is encouraged, and innovators are able to leverage the work done by others, standing on the shoulders of giants.

- **Standards.** The communication mode and content of a system. Examples would be a program's API, an XML schema, or TCP/IP.
- **Licensing.** What a user is permitted to do with a piece of software. A user may be restricted in use or even permitted full access to source code.

- **Innovation.** A community being able to collaborate and share ideas and software. This spans both the propensity for the software to encourage collaboration as well as actual collaboration happening within the using community.

3. Criteria

For each layer and each operation, criteria are established for assessing the openness of each operation on each layer with respect to standards, licensing, and community innovation. Unlike the Anvaari and Jansen model, which asks if it is possible to integrate, extend, or modify each layer, we ask how does the product's handling of standards, licensing, and innovation help to integrate, extend, or modify each layer.

This model uses categorical classifications to assess openness but avoids the value judgements embedded in the red, yellow, green color scheme in favor of green, yellow, blue.

a. Standards

In assessing how standards affect an operation on a layer, we must consider the kind of software and methods of integration that are offered, if any. In a software library, *open* might mean having integration take place by documented API calls that were developed in collaboration with many parties. A less open alternative might be documented API calls that are closed to changes aside from the framework's vendor. Not open at all might mean not being able to integrate or only through unpublished or private APIs. In contrast a web oriented framework might regard open as having integration take place over Hypertext Transfer Protocol (HTTP) with Extensible Markup Language (XML) data in a Simple Object Access Protocol (SOAP) message whose formatting is discovered in the Universal Description, Discovery, and Integration (UDDI) and which was published by the World Wide Web Consortium (W3C).

With respect to standards, the following classifications are used:

- I_s = The techniques for operating on a layer are based on documented, open standards that include community participation.

- II_s = The techniques for operating on a layer are partially standardized, or the standard is not subject to community participation.

- III_s = There are no techniques for operating on a layer or the techniques involve unsupported "hacks" or direct source code editing.

b. Licensing

In assessing how licensing applies to an operation on a layer, we consider permissions both for redistribution and access to different parts of the framework. The government has an interest in making sure it has access to all parts of a framework necessary and that models developed by one organization are not locked away from other government organizations—or even different projects in the same organization—due to licensing. We might expect to see less differentiation among the layers and operations here than in standards since software tends to be licensed *en masse*, not piece by piece.

Closed source software often has no provision for viewing its source code. Some vendors may charge a fee and require a non-disclosure agreement before allowing access to source code. Some vendors, whether open or closed source, make a good deal of sample code available to help developers at the application layer. Sometimes it is possible to integrate into the kernel of a framework through documented APIs (some open standards, II_s) but have no access to the kernel source code.

With respect to licensing, the following classifications are used:

- I_l = Users have the right to access, modify, and redistribute both their finished simulation and the framework's source code without express permission.

- II_l = Users are restricted in how they may access, modify, or redistribute either their finished simulation or the framework and its source code.

- III_l = Users may not redistribute their finished simulations or have no access to the framework's source code.

c. Innovation

In assessing how innovation applies to an operation on a layer, we consider both the encouragement offered to collaborate as well as the collaboration that is already happening. The government is interested in the benefits that many smaller acquisition programs can contribute to everyone, and with a thriving ecosystem with collaboration and shared ideas and products, the entire community is enriched.

Assessing a framework based on its community involvement may seem unfair to the framework, but the goal is not just to assess a piece of software on its own but rather how appropriate the framework is for use in acquisition simulation. This holistic approach centers on the needs of the user rather than on trying to isolate the framework from its intended audience. This also means that "involvement" must be defined within the proper scope. Niche markets would not be expected to have the same number of innovating participants as something with mass-market appeal. New frameworks with few users will need to be assessed based on the innovation demonstrated within the current user base.

With respect to innovation, the following classifications are used:

- I_i = Software and vendor lends itself to and a community takes advantage of innovation.
- II_i = Software or vendor may discourage or a community may not be greatly interested in innovation.
- III_i = Software or vendor restricts or there is no community of interest seeking innovation.

d. Summary of Criteria

The openness criteria is summarized in Table 4. For each issue, the criteria are applied to each layer and operation.

Table 4: Assessment criteria for the three openness three issues that are applied to the four layers and three operations.

Issue	Rating	Criteria
Standards	I_s	The techniques for operating on a layer are based on documented, open standards that include community participation.
	II_s	The techniques for operating on a layer are partially standardized, or the standard is not subject to community participation.
	III_s	There are no techniques for operating on a layer or the techniques involve unsupported "hacks" or direct source code editing.
Licensing	I_l	Users have the right to access, modify, and redistribute both their finished simulation and the framework's source code without express permission.
	II_l	Users are restricted in how they may access, modify, or redistribute either their finished simulation or the framework and its source code.
	III_l	Users may not redistribute their finished simulations or have no access to the framework's source code.
Innovation	I_i	Software and vendor lends itself to and a community takes advantage of innovation.
	II_i	Software or vendor may discourage or a community may not be greatly interested in innovation.
	III_i	Software or vendor restricts or there is no community of interest seeking innovation.

4. Model for Assessing Openness

Building on the model developed by Anvaari and Jansen (2010), we present a visual model that assesses openness both at a glance, in color and weight of markers, and in detail (Table 5). This follows the principles of small multiples, which encourages a visual cue to be repeated along axes and present both detailed information and information at a glance (Tufte, 2001).

The decision maker gains insight both in the process of applying the model and in examining the model. In applying the model, a deep and structured understanding of the models is developed. In examining the model, a decision maker may study the various aspects of proposed frameworks and consider where a less-open framework may be tolerated without adding unnecessary risk.

Table 5: A model for assessing the openness of simulation frameworks.

		Framework 1			Framework 2		
Layer	Operation	Std	Lic	Inn	Std	Lic	Inn
External Applications	Integrate	I_s	I_l	I_i	...		
	Extend	II_s	II_l	II_i			
	Modify	III_s	III_l	III_i			...
Internal Applications	Integrate	...					
	Extend						
	Modify						
Middleware	Integrate						
	Extend						
	Modify						
Kernel	Integrate						
	Extend						
	Modify						...

Std, s = Standards; Lic, l = Licensing; Inn, i = Innovation;
I, II, III are classifications.

5. Weights for User Assigned Value Systems

If a final numeric score is desired, weights can be assigned to the categories according to what layers, operations, or issues are most important. When summed, these weights provide for a tailored assessment of the frameworks. Here is where value can be applied, because the user specifies what is important.

Let L be the set of layers L = {External Applications, Internal Applications, Middleware, Kernel} and $\mathrm{l} \in \mathrm{L}$ be a layer. Let O be the set of operations O = {Integrate, Extend, Modify} and $\mathrm{o} \in \mathrm{O}$ be an operation. Let I be the set of issues {Standards, Licensing, Innovation} and $\mathrm{i} \in \mathrm{I}$ be an issue. Let $\mathrm{R}_{\mathrm{loi}}$ be the categorical rating assigned to a framework at the given layer, operation, and issue. Let $w_{\mathrm{loi}}(\mathrm{R}_{\mathrm{loi}})$ be the weighting function that returns the user assigned value for a given $\mathrm{R}_{\mathrm{loi}}$. Then the overall openness value V_{O} of a framework is given by Equation 1:

$$V_{\mathrm{O}} = \sum_{\mathrm{l} \in \mathrm{L}} \sum_{\mathrm{o} \in \mathrm{O}} \sum_{\mathrm{i} \in \mathrm{I}} w_{\mathrm{loi}}(\mathrm{R}_{\mathrm{loi}}) \tag{1}$$

The development of these weights for a particular use case is beyond the scope of this dissertation, but they may be used to help score frameworks against the specific needs of a program manager.

B. STUDY 1: ASSESSING OPENNESS

To demonstrate the feasibility of this assessment model, two simulation frameworks Delta3D and DMZ were selected and assessed using the methodology presented here. Of the hundreds of game engines available (M. Lewis & Jacobson, 2002), these two frameworks were selected because they represent two fundamentally different approaches to building visual simulation frameworks, often called game engines. They are also both readily available for download and inspection. The study demonstrates the feasibility of measuring openness to distinguish visual simulation architectures.

1. Methodology

After selecting the frameworks, the layers were identified and the criteria applied. Applying the model to the frameworks required an understanding of the frameworks involved. A careful study of the two frameworks preceded this study.

The layers of the software were defined for each framework. Specific tools, namespaces, folders, source code, and other pieces of the frameworks were identified. Clarity was required to ensure that the criteria were applied cleanly to the layers without one area bleeding over into another.

With the criteria laid out, each combination of layer, issue, and operation was examined. The criteria determined the categorical ratings to assign.

The ratings were compiled into a table, which is presented as a whole as well as divided into areas of interest, according to the distinctions made by the model.

2. Delta3D

Delta3D (Figure 4) is a successful open source game engine developed at the Naval Postgraduate School (NPS) in Monterey, CA. It has over six years and $1 million of development behind it. Its staff of programmers both maintain the framework and use the framework for research projects. It is also used by other organizations around the world. It boasts many features and integrates a number of open source libraries such as Open Scene Graph for rendering, Open Dynamics Engine for physics, and OpenAL for audio, to name a few.

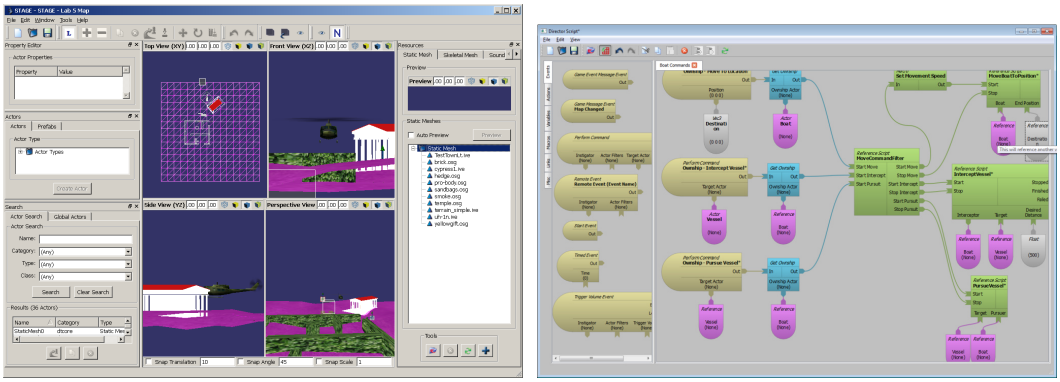
The Delta3D source consists of 160,000 lines of code and nearly 1,200 classes. It is written in C++. Table 6 lists the major namespaces into which Delta3D is divided and the size of each. The programming team uses good object-oriented programming practices. The code is representative of the conventional approach to game engine development. Much of the development team had prior experience with the Unreal engine by Epic Games, one of the largest game engines on the market. Consequently, Delta3D was developed following many Unreal paradigms. Classes are structured reasonably around the problem domain with the ubiquitous “Actor” class tying much of the functionality together.

Figure 4: Delta3D is an open source game engine used by many projects around the world and has a staff of programmers.

Table 6: Delta3D consists of many classes divided into namespaces related to their purpose.

Namespace	Classes	Lines of Code
dtABC	31	3,381
dtActors	98	10,616
dtAI	97	8,101
dtAnim	55	6,535
dtAudio	14	2,927
dtCore	155	24,204
dtDAL	111	13,397
dtDirector	98	15,080
dtDIS	16	1,589
dtGame	103	9,555
dtGUI	19	3,206
dtHLAGM	46	9,128
dtInspectorQt	21	2,243
dtLMS	13	711
dtNet	3	360
dtNetGM	11	1,971
dtQt	46	6,409
dtScript	1	57
dtTerrain	54	5,055
dtUtil	96	10,577
NA	57	1555
psGeodeTransform	1	18
sigslot	43	2090

Delta3D also comes with a number of "helper programs," which we call Internal Applications (Figure 5). One is called STAGE (Figure 5a), which helps in developing environments with buildings, terrain, actors, and more and can be thought of as a type of "level editor" for Delta3D. The objects described in STAGE can be manipulated programmatically by calling the various actors' functions. A new recently-released tool called Director (Figure 5b) provides a graphical environment for even non-programmers to script behaviors in a simulation.



(a) Delta3D's STAGE (b) Delta3D's Director

Figure 5: Delta3D has powerful tools to help build simulations such as STAGE for building environments and Director for scripting behaviors.

3. DMZ

DMZ (not an acronym) (Figure 6) is a new open source, component-based game engine developed at Naval Postgraduate School (NPS). It grew from a frustration that so many student projects could not be easily reintegrated into one source tree because of the fragility of the code. The classes and functionality developed by students touched too many other parts of the software causing a dependency quagmire. DMZ developer Randall Barker created a new game engine focused on developing small, reusable chunks of code that center around functionality and behavior rather than object encapsulation.

What he "invented" was service-oriented, component-based programming applied to visual simulation, though he explains it as just a natural engineering solution to the classic dependency problem he had always fought in OO programming. Note how well

this maps to the recommendation made by the Open Technology Development Roadmap Plan (Herz et al., 2006): "This report recommends shifts in the process of technology acquisition from closed, locked-in black box systems to open and modular approaches. These approaches are based on open standards, services based architectures, open source collaboration, and reference open source implementations."

Figure 6: DMZ is a new open source, component-based game engine used at NPS.

The DMZ source consists of over 98,000 lines of code and over 400 classes in over 800 files. It is written in C++, but simulations can also be constructed in the JavaScript or Lua scripting languages. Although the architecture is component-oriented, it is still built with classes and objects. All DMZ code resides in a single namespace. The developers instead group code in directories. Table 7 lists the major headings into which DMZ is divided and the size of each.

It is interesting and perhaps not surprising to observe that there are a greater number of *verb* class names than normally found in *object* oriented programming (Table 8). In Delta3D 5.5% of the classnames end in verbs, while in DMZ 9.7% of the classnames end in verbs. While not a scientific conclusion, this does suggest that DMZ is service (verb) oriented instead of object (noun) oriented.

DMZ does not have much in the way of Internal Applications, as it is not as mature as Delta3D and has a smaller staff of developers. There are some scripts that generate boilerplate code. These scripts probably contribute to some of the higher-than-expected

Table 7: DMZ consists of many classes divided by directory according to their purpose.

Directory	Classes	Lines of Code
frameworks/archive	11	2,530
frameworks/audio	11	1,962
frameworks/entity	28	6,849
frameworks/event	10	2,317
frameworks/input	18	2,659
frameworks/net	27	8,754
frameworks/object	27	5,727
frameworks/qt	73	14,185
frameworks/render	45	10,238
frameworks/weapon	6	1,288
foundation/libs	26	1,879
foundation/plugins	10	1,760
kernel/runtime	68	5,427
kernel/system	19	1,484
kernel/types	31	2,459

Table 8: The object-oriented Delta3D tends to be organized around nouns and adjectives, while the service-oriented DMZ tends to be organized around verbs.

Framework	Sample Class Names
Delta3D	BaseWater*Actor* AiActor*Registry* *Animatable* Sound*Command* Sky*Dome*
DMZ	EntityPlugin*Articulate* EntityPlugin*Follow* InputPluginChannel*Switch* ObjectPlugin*Highlight* ObjectPlugin*Select*

Weighted Methods per Class metrics since the scripts generate a number of placeholder methods that are often never used. Plans to create a "level editor" type of tool exist, but no such tool was available during this research for assessment. There is a "make" system called `lmk` for compiling components. The system is written in a scripting language called Lua (`lmk` = Lua make). This is used extensively in the development of DMZ simulations.

4. Identifying Software Layers

A clear delineation must be made for each of the four layers. Some layers will have a good deal of code behind them while others may be minimal or non-existent. Table 9 maps the four layers to specific parts of the frameworks.

Table 9: Layers for the two frameworks Delta3D and DMZ are broken out to aid in assessing openness.

Framework	Layer	Description
Delta3D	External Applications	The final simulations and applications built with the framework.
	Internal Applications	Tools stored in `utilities` folder (AIUtility, AnimationViewer, Exporters, GameStart, LMS, MapDump, ObjectViewer, ParticleEditor, STAGE).
	Middleware	Code not in the Kernel namespaces.
	Kernel	Code in the dtCore, dtGame, and dtDAL namespaces.
DMZ	External Applications	The final simulations and applications built with the framework.
	Internal Applications	Scripts in the `scripts` folder and the Lua "make" (`lmk`) system.
	Middleware	Code in the `frameworks` folder.
	Kernel	Code in the `foundation` and `kernel` folders.

5. Applying Criteria

The process of applying the criteria requires stepping through each combination of layer and operation to compare the three issues against the criteria established earlier.

a. External Applications

In both frameworks the openness of the the external applications is determined in large part by the developer of the final simulations, not the framework itself. Whether Extending, Integrating, or Modifying, a common theme is that there is little influence by the framework on what developers do.

Delta3D

With respect to Standards, developers Integrating, Extending, or Modifying another developer's External Applications are not guaranteed to have access via open standards. Delta3D neither requires nor forbids that developers use open standards, and there is no mechanism in the Delta3D architecture itself to enable it. Therefore, only a rating of II_{s} is appropriate for Delta3D for Integrating, Extending, or Modifying.

With respect to Licensing, developers working with Delta3D are bound by the Lesser GNU Public License (LGPL). This license does not require that developers using Delta3D to produce External Applications release these applications as open source. However, they are not forbidden from releasing these applications as open source either. Delta3D External Applications are rated II_{l} for licensing for Integrating, Extending, or Modifying.

With respect to Innovation, there is no provision in Delta3D to encourage developers of External Applications to seek collaboration and sharing within their applications. Although it is not prohibited, no evidence could be found of it occurring. Applications built with Delta3D are likely to be "one way" applications that are built once and not used by anyone else. Delta3D External Applications are rated $\mathrm{III}_{\mathrm{i}}$ for innovation for Integrating, Extending, or Modifying.

DMZ

With respect to Standards, developers Integrating, Extending, or Modifying External Applications have an advantage in DMZ. They can tap into the Object module, which manages the data defining the world of the simulation, or the Event module, which manages communication among modules and plugins. Even without published information for an External Application, the very architecture of DMZ enables developers to Integrate, Extend, or Modify applications in the same way regardless of the source. The standardized XML configuration files and the inspectable Object and Event modules aid developers in interacting with applications in a standard and straightforward way. Integrating, Extending, and Modifying DMZ External Applications are rated I_{s} for Standards.

With respect to Licensing, developers working with DMZ are bound by the Massachusetts Institute of Technology (MIT) License, one of the shortest of all open source licenses. It levies few requirements on developers except the need to give credit to the original DMZ developers and to hold DMZ blameless. Again, developers are not required to or forbidden from releasing External Applications as open source. Therefore, DMZ External Applications are rated II_{l} for licensing for Integrating, Extending, or Modifying.

With respect to Innovation, the very architecture of DMZ makes innovation of External Applications possible. Applications are made up of composable elements that can be shared and on which developers can collaborate. However, DMZ being a new development and mostly used in-house, the only evidence of innovation is in a very small community. DMZ External Applications are therefore rated II_{i} for innovation for Integrating, Extending, or Modifying.

b. Internal Applications

The characteristics of Internal Applications are directly influenced by the frameworks themselves. The nature and quality of the Internal Applications affect how developers are able to make use of the frameworks, and their openness can have lasting effects.

Delta3D

With respect to Standards, Delta3D has numerous tools to aid in the development of rich 3D worlds. They consist mostly of standalone tools, but they interoperate through shared data files. The STAGE tool for example supports the `.ive` file type, which comes from the open source Open Scene Graph (OSG) rendering engine. Unfortunately the `.ive` file type does not have a published specification, is defined only in the implementing code within OSG, and is now obsolete (OSGForum, 2011). The replacement file type `.osg` also has no published file format. Despite this, the community still uses `.osg` files as one of many standard Three Dimensional (3D) file formats.

Integrating with Delta3D Internal Applications consists primarily of loading data files, and as such Delta3D makes a good effort to support a number of standard, either open or *de facto*, file types and is rated I_s for Standards.

Extending or Modifying Delta3D Internal Applications is a different matter altogether. There is no approved technique for extending their behavior. The code is open source, fortunately, but that is a different issue. There is no plugin architecture or scripting or any technique (other than manipulating source code) for making these changes to Internal Applications. Extending or Modifying Delta3D Internal Applications are rated III_s for Standards.

With respect to Licensing, the Delta3D Internal Applications are released under the LGPL (or the GNU Public License (GPL) in the case of STAGE which uses the Qt library, also released under GPL). As such developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying Delta3D Internal Applications are rated I_l for Licensing.

With respect to Innovation, Delta3D Internal Applications do not have a community collaborating or sharing with them nor is there anything built-in to facilitate this. These are powerful and useful tools provided by the Delta3D development team but are final products themselves—not something with which to innovate a new solution. In-

tegrating, Extending, or Modifying Delta3D Internal Applications are rated III_i for Innovation.

DMZ

DMZ Internal Applications consist of a few scripts and the `lmk` build system. Integrating with these tools is as simple as command line calls. Therefore, Integrating with DMZ Internal Applications is rated I_s for Standards.

Other than editing the scripts or the `lmk` build system, there is no method for Extending or Modifying DMZ Internal Applications. Again, the code is open source, but that is a different issue, and since these Internal Applications are not themselves built with DMZ, which does provide a standard mechanism for changing behavior, Extending or Modifying DMZ Internal Applications are rated III_s for Standards.

With respect to Licensing, the DMZ Internal Applications are released under the MIT license. As such developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying DMZ Internal Applications are rated I_l for Licensing.

With respect to Innovation DMZ Internal Applications do not have a community collaborating or sharing with them nor is there anything built-in to facilitate this. These are scripts to aid in DMZ development, not something with which to innovate a new solution. Integrating, Extending, or Modifying DMZ Internal Applications are rated III_i for Innovation.

c. Middleware

The Middleware is the "meat" of the frameworks. These are the libraries, classes, plugins, and more that developers use in building their simulations. It is the Middleware that perhaps has the greatest influence on the day to day development of a simulation.

Delta3D

With respect to Standards, Delta3D Middleware can be Integrated through standard C++ function calls through published headers. The mechanism for integrating

with with the libraries is open, documented, and has some community participation. Integrating Delta3D Middleware is rated I_s for Standards.

Extending Delta3D Middleware could involve something as simple as subclassing a library class and using the new child class in place of the parent, or it could require changes to the original source code, as with Internal Applications. Extending Delta3D Middleware is rated II_s for Standards.

There is no approved technique for Modifying the Delta3D Middleware aside from editing the source code. Therefore, Modifying Delta3D Middleware is rated III_s for Standards.

With respect to Licensing, Delta3D Middleware is released under the LGPL. As such, developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying Delta3D Middleware is rated I_l for Licensing.

With respect to Innovation, Delta3D Middleware has some community involvement for collaboration and sharing, but the software is not particularly suited to users making unexpected and innovative uses of other people's work. Some such sharing goes on within the halls of NPS as students share code, but this does not achieve high innovation. Integrating Delta3D Middleware is rated II_i for Innovation.

Trying to Extend or Modify Delta3D Middleware is even less friendly to Maxwell's (2006) concept of Innovation. Extending and Modifying Delta3D Middleware is rated III_i for Innovation.

DMZ

With respect to Standards, DMZ Middleware can be Integrated through the standardized calls to the Object module or Event module, and these calls can be made with C++ or JavaScript. Further integration is possible with simple configuration of the XML files that define an application. Integrating DMZ Middleware is rated I_s for Standards.

Extending or Modifying DMZ middleware is possible through the same mechanism by which one Integrates. This is an architectural benefit of having loosely coupled components that communicate through restricted mechanisms, namely the Object

and Event modules. The XML configuration files permit easy swapping out of components making Modifying as normal an operation as Integrating, possibly more so. Extending and Modifying DMZ Middleware is also rated I_s for Standards.

With respect to Licensing, DMZ Middleware is released under the MIT license. As such, developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying DMZ Middleware is rated I_l for Licensing.

With respect to Innovation, DMZ Middleware Integration is well-suited to a collaborative and sharing environment. Component purpose and function are isolated, and there are simple, standardized techniques for composing applications. To date, there is not a DMZ community to speak of, except the development team that built and uses DMZ, and here power of innovation is used to good effect as one person's components are composed into another's project. Although the promise for good innovation is present, there is little community for proof. Integrating DMZ Middleware is rated II_i for Innovation.

Extending and Modifying DMZ Middleware enjoys the same promise of Innovation, and some proof of this is borne out in its development team. Component behavior can be Extended or Modified using the same technique as Integration, but as with Integration, there is little community for proof. Extending and Modifying DMZ Middleware is rated II_i for Innovation.

d. Kernel

The Kernel is the portion of the framework that "makes it tick." Developers do not generally need access to the Kernel for defining their simulations, though they likely will access it in some way in order to run their simulations.

Delta3D

With respect to Standards, the Delta3D Kernel can be Integrated through standard C++ function calls through published headers. As with the Middleware, the mechanism for integrating with the kernel is open, documented, and has some community participation. Integrating the Delta3D Kernel is rated I_s for Standards.

Extending the Delta3D Kernel can be achieved easily through inheritance or with more difficulty through editing source code. There are no standardized techniques or architectural allowances for enhancing the behavior of the Kernel. Extending the Delta3D Kernel is rated II_s for Standards.

There is no approved technique for Modifying the Delta3D Kernel aside from editing the source code. Therefore, Modifying the Delta3D Kernel is rated III_s for Standards.

With respect to Licensing the Delta3D Kernel is released under the LGPL. As such, developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying DMZ Middleware is rated I_l for Licensing.

With respect to Innovation, the Delta3D Kernel is situated in a similar way to the Middleware which, although it is possible to collaborate and share innovative solutions that Integrate with the Kernel, there is little community taking advantage of it. Integrating the Delta3D Kernel is rated II_i for Innovation.

Extending and Modifying the Delta3D Kernel is not well suited to innovation, and there is no community of activity there. Extending and Modifying the Delta3D Kernel is rated III_i for Innovation.

DMZ

With respect to Standards, the DMZ Kernel can be Integrated through standard C++ function calls through published headers like Delta3D, but unlike the rest of DMZ, the Kernel is not itself a component architecture (although some parts of the `foundation` directory are minor components). The DMZ Kernel is not well-documented, and this impairs a developer's ability to use it. It is instead the mechanism that loads components and enables their interconnections. Integrating the DMZ Kernel is rated II_s for Standards.

Extending or modifying the DMZ Kernel can only be accomplished through editing the source code. There is little room even for extension by inheritance. Extending and Modifying the DMZ Kernel is rated III_s for Standards.

With respect to Licensing, the DMZ Kernel is released under the MIT license. As such developers are not restricted in their use or redistribution. Integrating, Extending, or Modifying DMZ Middleware is rated I_{l} for Licensing.

With respect to Innovation, the DMZ Kernel can enjoy collaboration and sharing of innovative ways to Integrate the Kernel, but there is little community taking advantage of it. Integrating the DMZ Kernel is rated II_{i}.

Extending and Modifying the DMZ Kernel does not enjoy the same benefits as its Middleware, which is component based. Changing the Kernel requires editing its source code, and there is no community activity there. Extending and Modifying the DMZ Kernel is rated $\mathrm{III}_{\mathrm{i}}$ for Innovation.

6. Results

We gain insight by applying the criteria, and the results of the openness model applied to these two frameworks is summarized in Table 10. Although this example only shows us two frameworks of the many that may be assessed in the future, it demonstrates the usefulness of the model's contribution and its success in being able to differentiate between two visual simulation frameworks—two open source frameworks at that. Some observations may be made regarding the model.

Assigning notional weights to the weighting function allows for further analysis. These weights would be customized according to the needs of the program manager. Assume a weight of 1 for I, 2 for II, and 3 for III (Equation 2).

$$w_{\mathrm{loi}}(\mathrm{R}_{\mathrm{loi}}) = \begin{cases} 1, & \text{if } \mathrm{R}_{\mathrm{loi}} = \mathrm{I} \\ 2, & \text{if } \mathrm{R}_{\mathrm{loi}} = \mathrm{II} \\ 3, & \text{if } \mathrm{R}_{\mathrm{loi}} = \mathrm{III} \end{cases}, \forall\, \mathrm{l}, \mathrm{o}, \mathrm{i} \tag{2}$$

With a set of possible weights provided and focusing on the development operations, we can plot how the model differentiates between frameworks. Figures 7 and 8 show the results of plotting the weighted values as a stacked bar chart.

Table 10: Because Openness is more than licensing, these two example open source simulation frameworks are not equal with respect to openness.

		Delta3D			DMZ		
Layer	Operation	Std	Lic	Inn	Std	Lic	Inn
External Applications	Integrate	II_s	II_l	III_i	I_s	II_l	II_i
	Extend	II_s	II_l	III_i	I_s	II_l	II_i
	Modify	II_s	II_l	III_i	I_s	II_l	II_i
Internal Applications	Integrate	I_s	I_l	III_i	I_s	I_l	III_i
	Extend	III_s	I_l	III_i	III_s	I_l	III_i
	Modify	III_s	I_l	III_i	III_s	I_l	III_i
Middleware	Integrate	I_s	I_l	II_i	I_s	I_l	II_i
	Extend	II_s	I_l	III_i	I_s	I_l	II_i
	Modify	III_s	I_l	III_i	I_s	I_l	II_i
Kernel	Integrate	I_s	I_l	II_i	II_s	I_l	II_i
	Extend	II_s	I_l	III_i	III_s	I_l	III_i
	Modify	III_s	I_l	III_i	III_s	I_l	III_i

Std, s = Standards; Lic, l = Licensing; Inn, i = Innovation;
I, II, III are classifications.

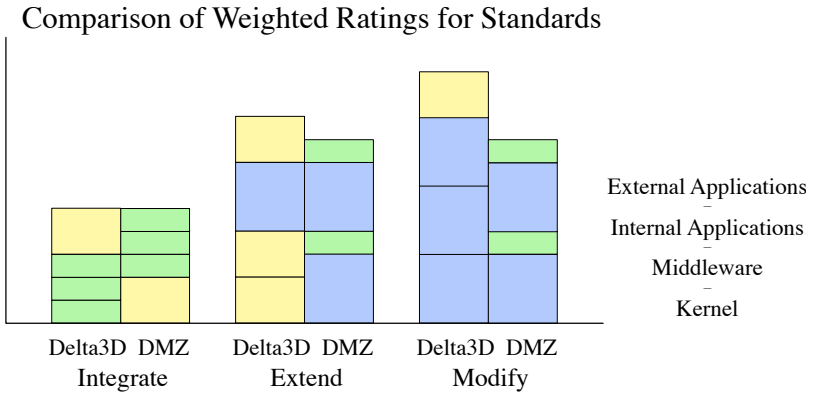


Figure 7: The two frameworks differ with respect to standards and development operation.

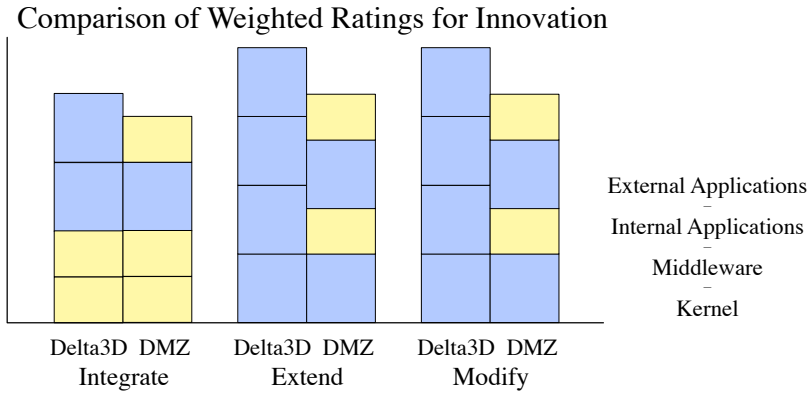


Figure 8: The two frameworks differ with respect to innovation and development operation.

7. Insights Gained

Not only does the model differentiate between the two frameworks, but the process of applying the model reveals insights. These insights, some examples of which are presented below, are an additional benefit of the model.

a. External Applications

One might remark that the External Applications section looks rather "boring" or has "low variability." This reminds us that External Applications are built by third parties, and that because of the latitude afforded by the licensing, developers using either framework may or may not generate open applications.

That the architecture of DMZ permits even third party applications to be manipulated in a standard fashion is a benefit that should not be overlooked. The government should be pleased that code that it pays for is open and accessible for use by other agencies.

b. Uniformity Across Operations

Although Delta3D has some variability across the operations Integrate, Extend, and Modify, DMZ shows little differentiation, especially in Middleware. This highlights a potential advantage of the component architecture in that it makes operations on the code more uniform. This should be seen as a big win for DMZ and possibly component

architectures in general. On the other hand Delta3D has consistency between Middleware and Kernel, whereas DMZ's Kernel is entirely different from its Middleware. Depending on the needs of the project, one may win out over the other.

c. Open Source License

It might come as a surprise to see a II_{ι} rating for both frameworks for External Applications since both frameworks are open source, but this highlights an interesting point about licensing. The licenses used for these frameworks do not require that External Applications be released as open source software. From one person's point of view, this may be a negative, because the work that a third party develops will not be accessible. From that third party's point of view, this may be desirable, because they are not forced to release their source code. The GPL is the classic example of a "viral" open source license that forces developers to release subsequent software as GPL. This would have been rated I_{ι} because of its strong insistence on open licensing, but whether or not it is a "good" thing is subjective.

C. SUMMARY

We have successfully shown that our openness model differentiates between two visual simulation frameworks and that we gain valuable insight in the process of applying and interpreting the model. The model contributes a new approach and tool for program managers and others to assess the nature of visual simulation frameworks with respect to openness.

Breaking out openness into more than just licensing, which is often what people think of when they hear "open," aided in the differentiation between frameworks. The standards issue in particular revealed architectural differences between the two frameworks we tested. Breaking out the operations revealed a valuable uniformity in interacting with one of the frameworks.

Weighting the categories according to layer, operation, and issue allows for customizing the model according to the values of a particular user. This also provides a numerical summary from the categorical data and may aid in analysis and decision support.

IV. REUSE

In this chapter, we develop and apply a model for assessing reuse based on the definition and need established in Chapter II: Literature Review. The model uses established software metrics and applies them in a new way in order to differentiate visual simulation frameworks based on their potential for reuse. These metrics are acknowledged to relate not only to reusability but also to general quality as well. We conduct a study in which we show that the model differentiates between two visual simulation frameworks.

A. DEVELOPING THE REUSE MODEL

We learned from Chapter II: Literature Review that there are internal and external attributes that affect reusability. Some external attributes that affect reusability are comprehensibility and maintainability. Some internal attributes that are related to these are complexity, coupling, and cohesion. We select relevant metrics for complexity, coupling, and cohesion and determine criteria for estimating transition points in the values of those metrics.

1. Complexity, Coupling, and Cohesion Metrics

Two major complexity metrics for procedural programming, McCabe (1976) and Halstead (1977), have survived as viable techniques for estimating complexity. The McCabe technique is still used in estimating complexity of methods within a class. This is used sometimes in the CK metric WMC, which is discussed in this chapter. Both techniques are often available in software development tools available to programmers.

McCabe (1976) describes a graph-theoretic, lexical, complexity measure that inspects the potential flow of execution through a program. Changes to a program flow resulting from such structures as `if` statements and `for` loops add to the complexity count. This metric is based on the decision structure of a program and is independent of the size, *e.g.*, adding a function does not increase the potential paths of execution.

McCabe defined the metric in terms of graph theory. The cyclomatic number $v(G)$ of a graph G with n vertices, e edges, and p connected components is

$$v(G) = e - n + p. \tag{3}$$

The examples given in Figure 9 help illustrate how the calculations are affected by some common control structures. In practice one often counts the predicates that are directly observable in source code and adds one, which McCabe proves to be equivalent to the graph-theoretic formulation.

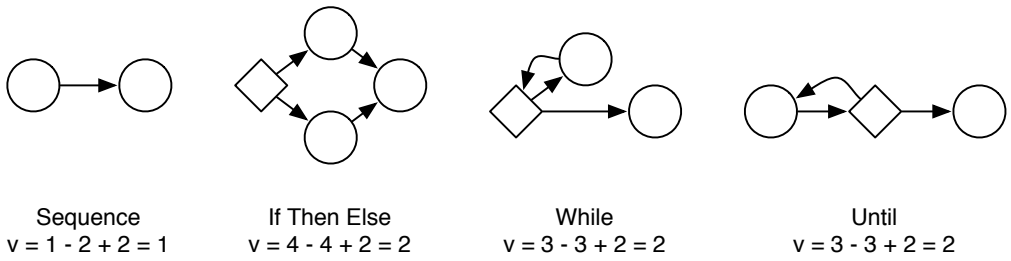


Figure 9: McCabe's cyclomatic complexity measures the possible paths of execution through a program.

McCabe found evidence that code with higher complexity values was less reliable and more "troublesome."

The CK suite of metrics are grounded in theory, relevant to OO programming, and validated empirically. Because of the importance of class design (Champeaux, Lea, & Faure, 1992), the CK metrics focus on measuring the complexity in the design of classes.

a. Weighted Methods per Class

Weighted Methods per Class (WMC) is a weighted sum of the count of methods in a class. S. R. Chidamber and Kemerer intentionally do not require how to weight the methods, suggesting that one could use a more traditional (procedural) complexity metric but that it is best left as an implementation decision by the practitioner. Li

and Henry (1993a) assume the use of McCabe's cyclomatic complexity. With a weight of 1 for all methods, the metric is simply the number of methods in each class.

To calculate WMC for a class A, there are at least two methods:

- Sum the chosen complexity metric for each method in class A. If m_{Ai} is method i of class A with n methods and $|m_{Ai}|$ is the chosen complexity metric for the method, then

$$\text{WMC}(A) = \sum_{i=0}^{n} |m_{Ai}|. \tag{4}$$

- Assume complexity of each method is the same, and count the number of methods in A. This technique is popular because of its simpler implementation and because there is disagreement over which complexity metric ought to be used.

They make several observations:

- The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

b. Depth of Inheritance Tree

Depth of Inheritance Tree (DIT) relates to scope and is a measure of how many ancestor classes could potentially affect this class.

To calculate DIT for class A one counts the number of parent classes one must traverse to reach the root. A root object such as `java.lang.Object` would have $\text{DIT} = 1$.

They make several observations:

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

c. Number of Children

Number of Children (NOC) is the number of immediate subclasses, which also relates to scope.

To calculate NOC for class A one counts the number of classes which directly inherit from A.

They make several observations:

- The greater the number of children, the greater the reuse, since inheritance is a form of reuse.
- The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing.
- The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

d. Coupling between Objects

Coupling Between Object Classes (CBO) relates to the interconnectedness of otherwise-unrelated (through inheritance) classes.

To calculate CBO for class A one counts the number of classes, beside A or ancestors of A, which are referenced either by instance variable or method call.

They make several observations:

- Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
- In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore, maintenance is more difficult.
- A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

The CBO metric has been the basis for many other coupling metrics. It is sometimes criticized (Hitz & Montazeri, 1995; Kitchenham, 2010) yet remains in use and validated by others (Basili et al., 1996; S. Chidamber et al., 1998).

e. Response for a Class

Response for a Class (RFC) is the number of methods that could possibly be called in response to a message being received.

To calculate RFC for class A one counts the number of methods in A and the number of methods that are called from within methods of A.

S. R. Chidamber and Kemerer and Li and Henry (1993a) do not specify whether or not this includes or excludes methods called to outside classes, but Hitz and Montazeri (1995) clarify RFC as the "union of the protocol a class offers to its clients and the protocols it requests from other classes."

S. R. Chidamber and Kemerer make several observations:

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

- The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

- A worst case value for possible responses will assist in appropriate allocation of testing time.

f. Lack of Cohesion in Methods

Lack of Cohesion in Methods (LCOM) relates to the degree of similarity, the difference among the instance variables used by each method in a class. This helps identify classes that may be trying to do too many things and whose behavior will be less predictable.

To calculate LCOM for class A, there are two similar but incompatible techniques that people use:

- Count the number of method pairs in A that do not use any of the same instance variables and subtract the number of method pairs in A that have at least one instance variable in common. Negative values are often reported as zero (Basili et al., 1996).

- Calculate the percentage of method pairs that do not use any of the same instance variables. This technique normalizes for the size of the class. Unfortunately the empirical validations that have been performed do not use this technique (Section 2)

They make several observations:

- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.

- Lack of cohesion implies classes should probably be split into two or more subclasses.

- Any measure of disparateness of methods helps identify flaws in the design of classes.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

These CK metrics have been examined, criticized, praised, and embraced by many researchers and practitioners, and are the metrics used in this research. However, for completeness, what follows are some other interesting metrics and research that has been done to improve the CK metrics. As so often happens with finer- and finer-grained improvements, the community that accepts them shrinks as well, and the original stands the test of time.

2. Empirical Evaluation of Metrics

Researchers have conducted studies to validate the CK metrics. These validations provide insight into what values are considered expected for the metrics although no literature discovered provided a thorough review of all empirical studies in order to generate consensus on these levels. The studies used metrics to predict fault-proneness as a surrogate for estimating software quality. Some literature reminds practitioners to use the metrics to guide further analysis when results are unexpected, and the same advice applies to the acquisition professional assessing simulation frameworks.

The empirical studies discovered in the literature cautioned users that no metric was a perfect predictor. However, these studies also confirmed that there was value in evaluating software against these metrics. For studies with significant findings for given metrics, a transition point from better to worse was calculated as one standard deviation above the mean.

Li and Henry (1993b) evaluated five of the CK metrics (not CBO) and their own metrics Message Passage Coupling (MPC) and Data Abstraction Coupling (DAC) to predict maintainability in a study of two commercial Ada applications built with Classic-Ada.™ They calculated WMC using the McCabe cyclomatic complexity technique, unlike the following studies which used the equal weighting method. Since the WMC metric dif-

fers in kind from the remaining studies, it is not reported here. Transition points for four metrics were calculated from the statistical results: DIT $\geqslant$ 2.7, NOC $\geqslant$ 1.6, RFC $\geqslant$ 75.6, LCOM $\geqslant$ 15.5.

Basili et al. (1996) evaluated eight information management systems against all six CK metrics. They found that the correlations among the metrics were weak, and that the relationship between CBO and RFC were most independent. They found that LCOM values were near zero in their systems and so did not provide any meaningful differentiation for their evaluation. Larger NOC values were better in their systems. Transition points for four metrics were calculated from the statistics results: WMC $\geqslant$ 28.3, DIT $\geqslant$ 3.3, CBO $\geqslant$ 14.4, RFC $\geqslant$ 67.3.

S. Chidamber et al. (1998) evaluated three software systems of an unnamed European bank. They found correlation among WMC, RFC, and CBO and suggest that subsets of the six CK metrics may be sufficient in assessing software. Transition points for six metrics were calculated from the statistical results: WMC $\geqslant$ 19.6, DIT $\geqslant$ 1.5, NOC $\geqslant$ 1.4, CBO $\geqslant$ 9.4, RFC $\geqslant$ 37.6, and LCOM $\geqslant$ 54.8.

Tang et al. (1999) evaluated a supervisory control and data acquisition system consisting of over 200 subsystems and 3 million lines of C++ code. They measured the six CK metrics and only validated WMC and RFC. Transition points for two metrics were calculated from the statistical results: WMC $\geqslant$ 19.6 and RFC $\geqslant$ 100.2.

Cartwright and Shepperd (2000) evaluated a system from a large European telecommunications company. The company employed more than 2,000 developers, and the system evaluated consisted of 133,000 lines of C++ source code. Of the CK metrics, they were only able to calculate DIT and NOC and found that the maximum values encountered were two and four respectively. Standard deviations were not reported. Because of the low values for these metrics, they only offered subjective evaluations but noted that the two CK metrics that they used could point out problematic classes.

Olague et al. (2007) evaluated the open source Mozilla project Rhino, a 100% Java implementation of JavaScript. Rhino is considered an example of agile software develop-

ment based on the principles of the Agile Alliance (Olague et al., 2007; Agile Alliance, 2011). They evaluated six releases of the source code from v1.4R3 to v1.5R5, which consists of 148 classes and 36,000 lines of code. They used an alternate cohesion metric, so their results are not comparable here. Transition points for five metrics were calculated from the statistical results: WMC $\geqslant$ 28.9, DIT $\geqslant$ 1.4, NOC $\geqslant$ 1.5, CBO $\geqslant$ 6.4, and RFC $\geqslant$ 32.7.

These empirical studies can be summarized to form a loose understanding of transition points from better to worse values for metrics (Table 11). These values should be taken with caution. Since our purpose for them is to provide insight into assessing software, not to predict hard outcomes, they still are useful.

Table 11: Transition points identified in empirical validation studies in the literature.

Paper	WMC	DIT	NOC	CBO	RFC
Li & Henry, 1993b	-	2.7	1.6	-	75.6
Basili et al., 1996	28.3	**3.3**	-	**14.4**	67.3
S. Chidamber et al., 1998	**19.6**	1.5	**1.4**	9.4	37.6
Tang et al., 1999	19.6	-	-	-	**100.2**
Cartwright & Shepperd, 2000	-	2.0	**4.0**	-	-
Olague et al., 2007	**28.9**	**1.4**	1.5	**6.4**	**32.7**

Boldface indicates chosen transition points.

There is some variation among the values observed in the validation experiments, and this seems to be accepted in the literature. Therefore, it is only through our summarizing the body of studies that transition points are identified.

3. Criteria for Metrics

There is no consensus in the literature for precise divisions in the metrics regarding "good" and "bad" values. However, ranges can be inferred from normal to higher than normal by examining the statistical results of several empirical studies and adding one standard deviation to the mean. This is chosen as a conservative estimator. The one sided

Tchebysheff inequality (Equation 5) tells us that the mean plus one standard deviation will always contain both the mean and median, regardless of the underlying distribution (Equation 6). Since too little data is reported in the literature to provide a more aggressive dividing line, μ plus $k = 1$ standard deviations is a reasonable choice. Three levels are defined here for the six CK metrics. These levels are based on the low and high estimates discovered in empirical studies.

$$Pr(X - \mu \geqslant k\sigma) \leqslant \frac{1}{1 + k^2} \tag{5}$$

$$|\mu - m| \leqslant \sigma \tag{6}$$

a. Weighted Method Complexity

When using equal complexity weights among methods (counting the methods in a class), the lowest published level that we infer as a transition is 19.6 (S. Chidamber et al., 1998; Tang et al., 1999). The highest level is 28.9 (Olague et al., 2007). Therefore, we divide the criteria into three categories: I_{WMC} for WMC = [0,19.6), II_{WMC} for WMC = [19.6,28.9), and III_{WMC} for WMC = [28.9,∞).

b. Depth of Inheritance Tree

Empirical studies found that many projects had very low values of DIT. The lowest published level that we infer as a transition is 1.4 (Olague et al., 2007), and the highest level is 3.3 (Basili et al., 1996). We divide the metric into three categories: I_{DIT} for DIT = [0,1.4), II_{DIT} for DIT = [1.4,3.3), and III_{DIT} for DIT = [3.3,∞).

c. Number of Children

Empirical studies suggested that a high number of children could indicate unnecessary inheritance and thus induce inheritance coupling. The lowest published level that we infer as a transition is 1.5 (S. Chidamber et al., 1998), and the highest level is 4.0

(Cartwright & Shepperd, 2000). We divide the metric into three categories: I_{NOC} for NOC = [0,1.4), II_{NOC} for NOC = [1.4,4.0), and III_{NOC} for NOC = [4.0,∞).

d. Coupling between Objects

Although there have been many follow-on metrics related to coupling, CBO remains a meaningful metric. The lowest published level that we infer as a transition is 6.4 (Olague et al., 2007), and the highest level is 14.4 (Basili et al., 1996). We divide the metric into three categories: I_{CBO} for CBO = [0,6.4), II_{CBO} for CBO = [6.4,14.4), and III_{CBO} for CBO = [14.4,∞).

e. Response Set for a Class

Studies confirmed the effectiveness of the RFC metric. The lowest published level that we infer as a transition is 32.7 (Olague et al., 2007), and the highest level is 100.2 (Tang et al., 1999). We divide the metric into three categories: I_{RFC} for RFC = [0,32.7), II_{RFC} for RFC = [32.7,100.2), and III_{RFC} for RFC = [100.2,∞).

f. Lack of Cohesion in Methods

There have also been many follow-on metrics to LCOM, but it stands as a useful metric. Unfortunately the version of LCOM that is reported in the studies is not the percentage method, which is normalized for class size, but rather the count of method pairs that do not share a common instance variable minus the count of method pairs that do. Because we do not have published LCOM values that we can apply, we strike LCOM from our list.

g. Summary of Reuse Criteria

Criteria for these metrics vary in the literature, but they can be sifted and summarized to provide some level of insight into the software being assessed. Table 12 summarizes the criteria used in this research to represent three quality levels.

Table 12: A thorough review of empirical validation studies provides transition points to use as criteria with software metrics.

	Lower Bounds		
Metric	I	II	III
Weighted Methods per Class (WMC)	0	19.6	28.9
Depth of Inheritance Tree (DIT)	0	1.4	3.3
Number of Children (NOC)	0	1.4	4.0
Coupling Between Object Classes (CBO)	0	6.4	14.4
Response for a Class (RFC)	0	32.7	100.2

4. Model for Assessing Reuse

To apply the model to the software frameworks, we calculate the metrics using a software analysis tool and apply the criteria to populate the model.

For continuity with the openness model and to aid in understanding different portions of the code, the criteria are applied to the two layers Middleware and Kernel. It should be expected that External and Internal Applications are assessed, since they represent simulations built or tools included *with* the framework, not the framework itself. Clear lines must be drawn with each framework regarding which code belongs in which layer.

The data may be presented such as in Table 13 where all of the ratings are presented at once, and both a detailed inspection and a high-level glance can provide insight.

5. Weights for User Assigned Value Systems

If a final numeric score is desired, weights can be assigned to the categories according to what layer and metric are most important. When summed, these weights provide for a tailored assessment of the frameworks.

Let L be the set of layers $\mathrm{L} = \{\text{Middleware, Kernel}\}$ and $\mathrm{l} \in \mathrm{L}$ be a layer. Let M be the set of metrics $\{\text{WMC, DIT, NOC, CBO, RFC}\}$ and $\mathrm{m} \in \mathrm{M}$ be a metric. Let R_{lm} be the categorical rating assigned to a framework at the given layer and metric. Let $w_{\mathrm{lm}}(\mathrm{R}_{\mathrm{lm}})$

Table 13: A model for presenting metrics for simulation frameworks help identify potential risks to reusability.

Code	Classes	WMC	DIT	NOC	CBO	RFC
Delta3D	x,xxx	I	...			
Middleware	...	II				
Kernel		III				
DMZ		...				
Middleware						
Kernel						

be the weighting function that returns the user assigned value for a given R_{lm}. Then the overall openness value V_R of a framework is given by Equation 7:

$$V_R = \sum_{l \in L} \sum_{m \in M} w_{lm}(R_{lm}) \tag{7}$$

The development of these weights for a particular use case is beyond the scope of this dissertation, but they may be used to help score frameworks against the specific needs of a program manager.

B. STUDY 2: ASSESSING REUSE

To demonstrate the feasibility of this assessment model, two simulation frameworks Delta3D and DMZ (described in Sections 2 and 3) were analyzed with the reuse model. To calculate the metrics, we used *Understand* from Scientific Toolworks, Inc. This software has a Perl API for accessing the lexical database created by a software project. Through this API one can write scripts to collect metrics that are not built-in to the software itself, such as the scripts used to collect CK metrics here.

1. Methodology

The first step in the reuse study was loading the two frameworks into the analysis tool *Understand*. This tool parsed all the source files in the two projects and created a lexical database that could be queried later.

A number of shell, awk, and Perl scripts (Appendix A) were used to extract the necessary metrics. Some metrics were generated by the reporting function within *Understand* itself, and some required accessing the lexical database with the API through Perl.

The metrics were compiled in spreadsheets and keyed by class or filename, as appropriate. From the spreadsheets, the data could be aggregated by framework, layer, or other division. The ratings were assigned within the spreadsheet by formulas based on the transition points developed from the literature.

2. Calculating the Metrics

The two codebases were analyzed to calculate CK metrics. The data was aggregated by namespace for Delta3D and major directory for DMZ and then aggregated again according to the software layers defined earlier: Middleware and Kernel. A detailed listing of the results is given in Table 14 for Delta3D and Table 15 for DMZ. A summary at the Middleware and Kernel levels is given in Table 16.

3. Applying the Criteria

We can now apply the criteria provided in the previous chapter to these metrics. Each metric value is compared against the transition points to determine the rating. The data was aggregated by namespace for Delta3D and major directory for DMZ and then aggregated again according to the software layers defined earlier: Middleware and Kernel. A detailed listing of the resulting ratings is given in Table 17 for Delta3D and Table 18 for DMZ. A summary is given in Table 19. Subscripts on the categories, *e.g.*, WMC in I_{WMC}, are left off to improve readability in the densely-packed tables.

Table 14: Detailed listing of CK metrics for the Delta3D framework

Code	Classes	WMC	DIT	NOC	CBO	RFC
Delta3D	1,191	12.43	1.96	0.63	7.43	38.84
Middleware	822	11.51	1.86	0.49	7.32	39.64
dtABC	31	14.58	2.77	0.42	9.45	56.42
dtActors	98	9.72	4.39	0.30	7.68	102.63
dtAI	97	10.81	0.93	0.33	5.33	18.09
dtAnim	55	10.91	1.60	0.15	9.11	29.22
dtAudio	14	18.86	3.07	0.00	7.79	67.79
dtDirector	98	15.93	2.51	0.60	13.20	65.09
dtDIS	16	7.81	0.69	0.13	10.25	10.25
dtGUI	19	14.79	1.32	0.11	9.74	22.37
dtHLAGM	46	13.67	1.13	0.24	6.43	23.93
dtInputISense	1	11.00	4.00	0.00	7.00	59.00
dtInputPLIB	1	12.00	4.00	0.00	6.00	60.00
dtInspectorQt	21	16.71	1.95	0.86	9.14	22.10
dtLMS	13	7.31	1.00	0.00	4.15	18.85
dtNet	3	12.67	1.67	0.00	13.67	19.67
dtNetGM	11	16.82	2.27	0.18	11.91	42.55
dtQt	46	19.17	1.63	0.46	13.59	49.22
dtScript	1	11.00	3.00	0.00	4.00	32.00
dtTerrain	54	7.83	1.39	0.11	5.22	22.41
dtUtil	96	11.07	0.66	1.64	3.43	14.56
NA	57	3.82	1.56	0.32	2.07	29.93
psGeodeTransform	1	1.00	1.00	0.00	6.00	1.00
sigslot	43	6.00	1.30	0.65	2.51	9.95
Kernel	369	14.48	2.20	0.93	7.69	37.08
dtCore	155	20.70	2.15	1.01	8.80	45.49
dtDAL	111	8.95	2.61	1.03	7.30	30.14
dtGame	103	11.10	1.84	0.69	6.46	31.90

Table 15: Detailed listing of CK metrics for the DMZ framework

Code	Classes	WMC	DIT	NOC	CBO	RFC
DMZ	475	13.39	0.92	1.19	8.37	41.81
Middleware	319	13.52	1.17	0.77	10.24	52.82
frameworks/archive	11	15.91	1.18	0.82	10.82	53.73
frameworks/audio	11	15.18	0.82	0.27	10.00	42.64
frameworks/entity	28	12.96	1.89	0.04	11.39	138.00
frameworks/event	10	23.20	0.60	1.20	9.50	42.70
frameworks/input	18	16.56	1.00	2.11	10.06	43.11
frameworks/net	71	6.08	1.07	0.80	7.17	19.08
frameworks/object	27	20.04	1.26	2.96	10.00	73.67
frameworks/qt	80	15.88	1.24	0.30	11.45	52.75
frameworks/render	57	13.19	0.95	0.37	11.19	42.51
frameworks/weapon	6	14.00	1.67	0.00	18.33	122.67
Kernel	156	13.12	0.43	2.06	4.54	19.31
foundation/libs	26	11.73	0.42	0.38	3.19	16.85
foundation/plugins	10	9.40	1.00	0.00	7.90	31.40
kernel/runtime	70	11.29	0.53	4.06	6.51	17.69
kernel/system	19	15.21	0.37	1.32	1.47	23.05
kernel/types	31	18.32	0.06	0.10	2.03	18.84

Table 16: Summary listing of CK metrics for Delta3D and DMZ frameworks

Code	Classes	WMC	DIT	NOC	CBO	RFC
Delta3D	1,191	12.43	1.96	0.63	7.43	38.84
Middleware	822	11.51	1.86	0.49	7.32	39.64
Kernel	369	14.48	2.20	0.93	7.69	37.08
DMZ	475	13.39	0.92	1.19	8.37	41.81
Middleware	319	13.52	1.17	0.77	10.24	52.82
Kernel	156	13.12	0.43	2.06	4.54	19.31

Table 17: Detailed listing of reuse ratings for the Delta3D framework

Code	Classes	WMC	DIT	NOC	CBO	RFC
Delta3D	1,191	I	II	I	II	II
Middleware	822	I	II	I	II	II
dtABC	31	I	II	I	II	II
dtActors	98	I	III	I	II	III
dtAI	97	I	I	I	I	I
dtAnim	55	I	II	I	II	I
dtAudio	14	I	II	I	II	II
dtDirector	98	I	II	I	II	II
dtDIS	16	I	I	I	II	I
dtGUI	19	I	I	I	II	I
dtHLAGM	46	I	I	I	II	I
dtInputISense	1	I	III	I	II	II
dtInputPLIB	1	I	III	I	I	II
dtInspectorQt	21	I	II	I	II	I
dtLMS	13	I	I	I	I	I
dtNet	3	I	II	I	II	I
dtNetGM	11	I	II	I	II	II
dtQt	46	I	II	I	II	II
dtScript	1	I	II	I	I	I
dtTerrain	54	I	I	I	I	I
dtUtil	96	I	I	II	I	I
NA	57	I	II	I	I	I
psGeodeTransform	1	I	I	I	I	I
sigslot	43	I	I	I	I	I
Kernel	369	I	II	I	II	II
dtCore	155	II	II	I	II	II
dtDAL	111	I	II	I	II	I
dtGame	103	I	II	I	II	I

Table 18: Detailed listing of reuse ratings for the DMZ framework

Code	Classes	WMC	DIT	NOC	CBO	RFC
DMZ	475	I	I	I	II	II
Middleware	319	I	I	I	II	II
frameworks/archive	11	I	I	I	II	II
frameworks/audio	11	I	I	I	II	II
frameworks/entity	28	I	II	I	II	III
frameworks/event	10	II	I	I	II	II
frameworks/input	18	I	I	II	II	II
frameworks/net	71	I	I	I	II	I
frameworks/object	27	II	I	II	II	II
frameworks/qt	80	I	I	I	II	II
frameworks/render	57	I	I	I	II	II
frameworks/weapon	6	I	II	I	III	III
Kernel	156	I	I	II	I	I
foundation/libs	26	I	I	I	I	I
foundation/plugins	10	I	I	I	II	I
kernel/runtime	70	I	I	III	II	I
kernel/system	19	I	I	I	I	I
kernel/types	31	I	I	I	I	I

Table 19: Summary listing of reuse ratings for Delta3D and DMZ frameworks

Code	Classes	WMC	DIT	NOC	CBO	RFC
Delta3D	1,191	I	II	I	II	II
Middleware	822	I	II	I	II	II
Kernel	369	I	II	I	II	II
DMZ	475	I	I	I	II	II
Middleware	319	I	I	I	II	II
Kernel	156	I	I	II	I	I

4. Results

The results of the reuse ratings and the process of interpreting and studying the results provides valuable insight into the software being assessed. This helps the acquisition professional better understand the frameworks under consideration to make a more informed decision.

To aid in the comparison between Delta3D and DMZ, we normalize the DMZ number of classes according to Delta3D's size. To adjust for the fewer classes in DMZ (about 60% fewer), we solve for x in the relationship given by Equation 8 where M is the number of DMZ classes in a given subset, and N_i is the number of classes in each framework with $i \in \{DMZ, Delta3D\}$.

$$\frac{M}{N_{DMZ}} = \frac{x}{N_{Delta3D}} \tag{8}$$

For example, if we counted 24 classes in DMZ and 70 classes in Delta3D for some grouping, then to find the adjusted DMZ value x that can be compared to 70, we take the number of DMZ classes in the grouping $M = 50$, the total number of classes in Delta3D $N_{Delta3D} = 1191$, the total number of classes in DMZ $N_{DMZ} = 475$, and compute x as shown in Equation 9:

$$\begin{aligned} \frac{M}{N_{DMZ}} &= \frac{x}{N_{Delta3D}} \\ x &= N_{Delta3D}\frac{M}{N_{DMZ}} \\ x &= 1191\frac{24}{475} \\ x &= 60.2 \end{aligned} \tag{9}$$

We can then compare the adjusted DMZ value 60.2 to the Delta3D value 70. When this adjustment is made, it will be noted in the table or figure that accompanies it.

Assigning notional weights to the weighting function allows for further analysis. These weights would be customized according to the needs of the program manager. Assume a weight of 1 for I, 2 for II, and 3 for III (Equation 10).

$$w_{lm}(R_{lm}) = \begin{cases} 1, & \text{if } R_{lm} = \text{I} \\ 2, & \text{if } R_{lm} = \text{II} \\ 3, & \text{if } R_{lm} = \text{III} \end{cases}, \forall\, l, m \tag{10}$$

With a set of possible weights provided and focusing on the development operations, we can plot how the model differentiates between frameworks. Figure 10 shows the results of plotting the weighted values as a stacked bar chart.

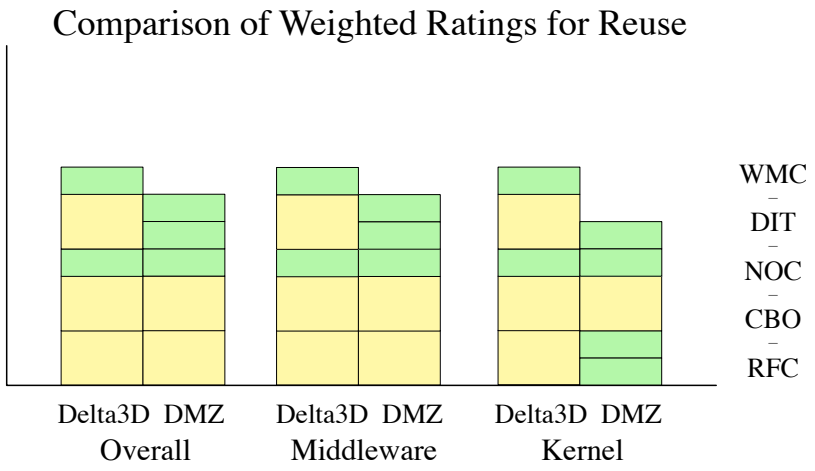


Figure 10: The two frameworks differ with respect to reuse.

a. Weighted Methods per Class

Both frameworks are rated I_{WMC} for WMC, which counts the number of methods in each class, revealing that in general both frameworks have reasonably-sized classes that are not too complex. Plotting the number of classes against their WMC values on a logarithmic scale (Figure 11) reveals that DMZ has a slightly higher average WMC than Delta3D. It is interesting to take a look at the top ten "hot spot" classes with respect to

WMC (Table 20). The first two classes `dtCore::RefPtr` and `dtCore::ObserverPtr` inherit from Open Scene Graph classes and perhaps could be excused. Despite the fact that DMZ has a higher average WMC, only two DMZ classes make it into the top ten (the next one is ranked #18). This may be another result of the component approach to development which values many, same-sized, composable components over "kitchen-sink" objects.

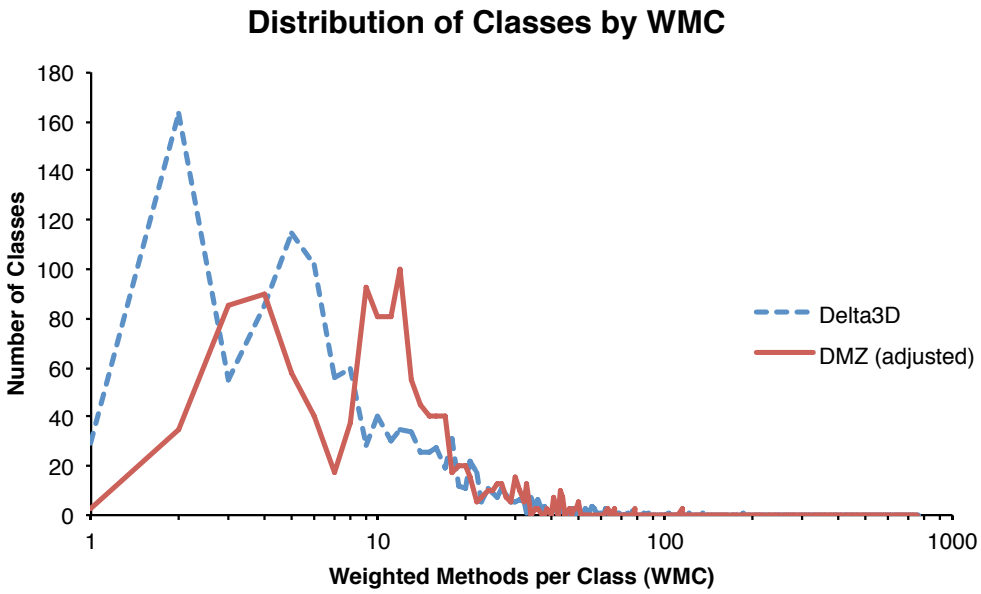


Figure 11: Number of classes per framework plotted by WMC reveals that DMZ has a higher average WMC.

b. Depth of Inheritance Tree

Delta3D is rated $\mathrm{II}_{\mathrm{DIT}}$, and DMZ is rated $\mathrm{I}_{\mathrm{DIT}}$ for DIT. It is expected that component based architectures are "flatter" with respect to inheritance (Qingqing & Xinke, 2009), and the findings here are consistent with that observation. Lower DIT values indicate that classes tend to be near the top of the inheritance chain and that inheritance is not the primary means by which functionality is extended. Recall that lower DIT values suggest lower dependencies on other classes and thus more loosely coupled code.

Table 20: The top ten classes with the highest WMC values reveals more potentially-problematic classes in Delta3D.

Rank	Class	Framework	Layer	WMC
1	dtCore::RefPtr	Delta3D	Kernel	763
2	dtCore::ObserverPtr	Delta3D	Kernel	186
3	dtGame::GameManager	Delta3D	Kernel	135
4	dtGame::DeadReckoningHelper	Delta3D	Kernel	121
5	dmz::ObjectModuleBasic	DMZ	Middleware	114
6	dtUtil::DataStream	Delta3D	Middleware	102
7	dtHLAGM::HLAComponent	Delta3D	Middleware	87
8	dtUtil::KDTree	Delta3D	Middleware	84
9	dmz::ObjectModule	DMZ	Middleware	78
10	dtAnim::Cal3DModelWrapper	Delta3D	Middleware	77

Figure 12 reveals that Delta3D has a greater number of classes with high DIT values than DMZ. Table 21 lists the top ten classes ranked by DIT values as well as the highest-ranked DMZ class, which arrives after the top 380 classes with DIT values [3..8]. Between the two frameworks there are 289 classes with DIT = 2.

c. Number of Children

Both frameworks are rated I_{NOC} for NOC indicating good levels of inheritance. Recall from Section 1 that some inheritance suggests good reuse but that greater values of NOC may suggest improper abstraction of classes (S. R. Chidamber & Kemerer, 1994).

Plotting the number of classes against their NOC values on a logarithmic scale (Figure 13) reveals that Delta3D and DMZ are fairly matched in NOC with 84.0% of the Delta3D classes and 82.7% of the DMZ classes at NOC = 0. A look at the top ten classes ranked by NOC (Table 22) reveals a spike in the `dmz::plugin` class with a whopping 167 children—almost double the number two class. The high NOC count for `dmz::plugin` reminds us that in a component architecture we expect to see all of the components derive from the base component (plugin) class, not from other components

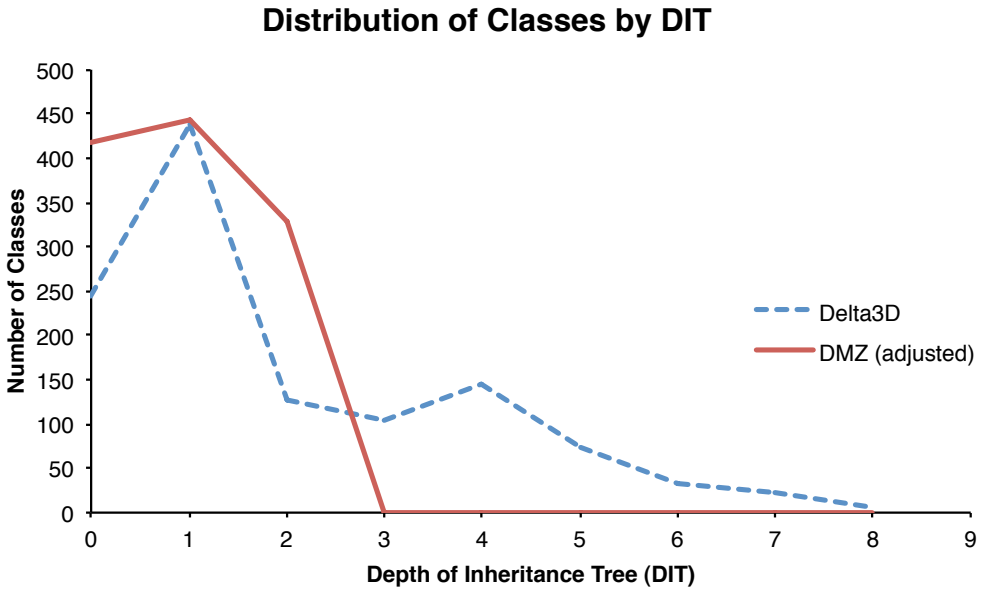


Figure 12: Number of classes per framework plotted by DIT reveals that Delta3D has a greater number of classes with high DIT values.

Table 21: The top ten classes with the highest DIT values reveals more deep-inheritance classes in Delta3D.

Rank	Class	Framework	Layer	DIT
1	dtActors::WaterGridActor	Delta3D	Middleware	8
2	dtActors::WeatherEnvironmentActor	Delta3D	Middleware	8
3	dtActors::TaskActorGameEvent	Delta3D	Middleware	8
4	dtActors::SkyDomeEnvironmentActor	Delta3D	Middleware	8
5	dtActors::TaskActorOrdered	Delta3D	Middleware	8
6	dtActors::TaskActorRollup	Delta3D	Middleware	8
7	dtAnim::Cal3DGameActor	Delta3D	Middleware	7
8	dtActors::DirectorActor	Delta3D	Middleware	7
9	dtActors::WaterGridActorProxy	Delta3D	Middleware	7
10	dtActors::DistanceSensorActor	Delta3D	Middleware	7
...				
380	sigslot::signal8	Delta3D	Middleware	3
381	dmz::RenderPluginEventOSG	DMZ	Middleware	2

that have been extended. This is the proper behavior for a component based architecture, and the metrics confirm that DMZ matches this behavior.

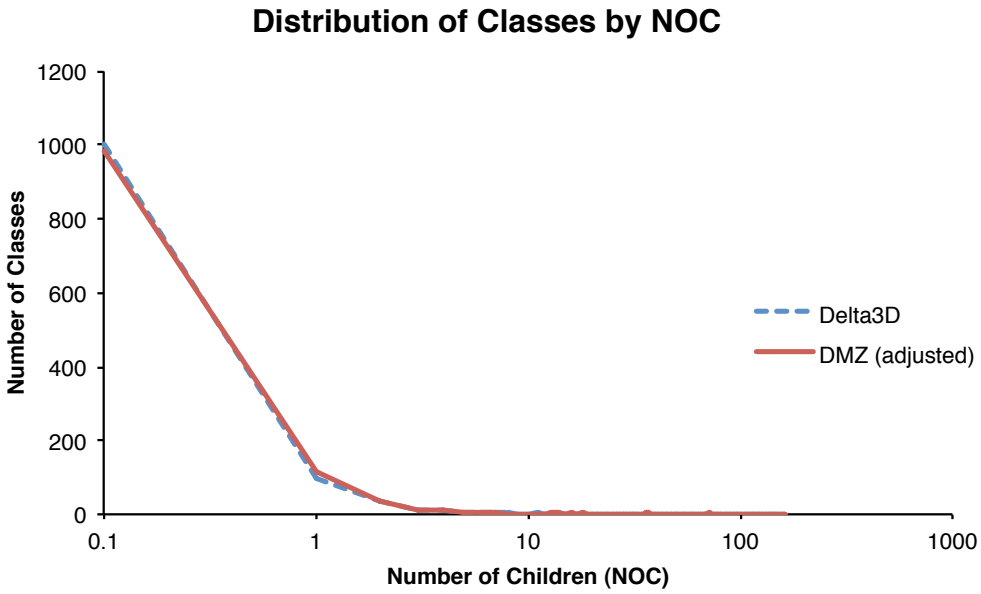


Figure 13: Number of classes per framework plotted by NOC reveals that Delta3D and DMZ are fairly matched in NOC values.

d. Coupling between Objects

Both frameworks are rated $\mathrm{II}_{\mathrm{CBO}}$ for CBO. In Table 19 we observe that Delta3D scores $\mathrm{II}_{\mathrm{CBO}}$ for both Kernel and Middleware, while DMZ scores different between Kernel and Middleware, $\mathrm{I}_{\mathrm{CBO}}$ and $\mathrm{II}_{\mathrm{CBO}}$, respectively. This might not surprise us since we know that the DMZ Kernel is fundamentally different than its Middleware. Recall that a higher CBO value suggests greater complexity, greater interdependence, and greater fragility when reusing code.

Plotting the number of classes against their CBO values on a logarithmic scale (Figure 14) reveals that Delta3D has a greater number of low-CBO classes and that CBO values are fairly evenly distributed across DMZ. However, it is interesting to take a

Table 22: The top ten classes with the highest NOC values reveals a close matching between Delta3D and DMZ.

Rank	Class	Framework	Layer	NOC
1	dmz::Plugin	DMZ	Kernel	167
2	dtUtil::Enumeration	Delta3D	Middleware	86
3	dmz::ObjectObserverUtil	DMZ	Middleware	71
4	dmz::TimeSlice	DMZ	Kernel	60
5	dtUtil::Exception	Delta3D	Middleware	57
6	dmz::MessageObserver	DMZ	Kernel	37
7	dmz::InputObserverUtil	DMZ	Middleware	36
8	dtCore::Base	Delta3D	Kernel	34
9	dtCore::Transformable	Delta3D	Kernel	25
10	dtDirector::ActionNode	Delta3D	Middleware	23

look at the top ten classes ranked by CBO. Table 23 reveals that Delta3D also has some of the highest-CBO values. The next DMZ class is ranked #16.

e. Response for a Class

Both frameworks are rated $\mathrm{II}_{\mathrm{RFC}}$ for RFC, and again we observe that the DMZ Kernel is rated slightly different at $\mathrm{I}_{\mathrm{RFC}}$.

Plotting the number of classes against their RFC values on a logarithmic scale (Figure 15) reveals that Delta3D has a number of very small, almost empty, classes and that DMZ otherwise has lower RFC values. Table 24 lists the top eleven classes ranked by RFC. The eleventh is added to the table because it is the first appearance of a DMZ class. Recall that a higher RFC value suggests greater complexity, greater potential for unintended consequences, and greater fragility when reusing code.

f. Top 100 Classes

Looking at the top 100 classes sorted in turn by each metric in both frameworks shows that DMZ has a smaller presence in these potential hot spot classes than Delta3D (Table 25 and Figure 16).

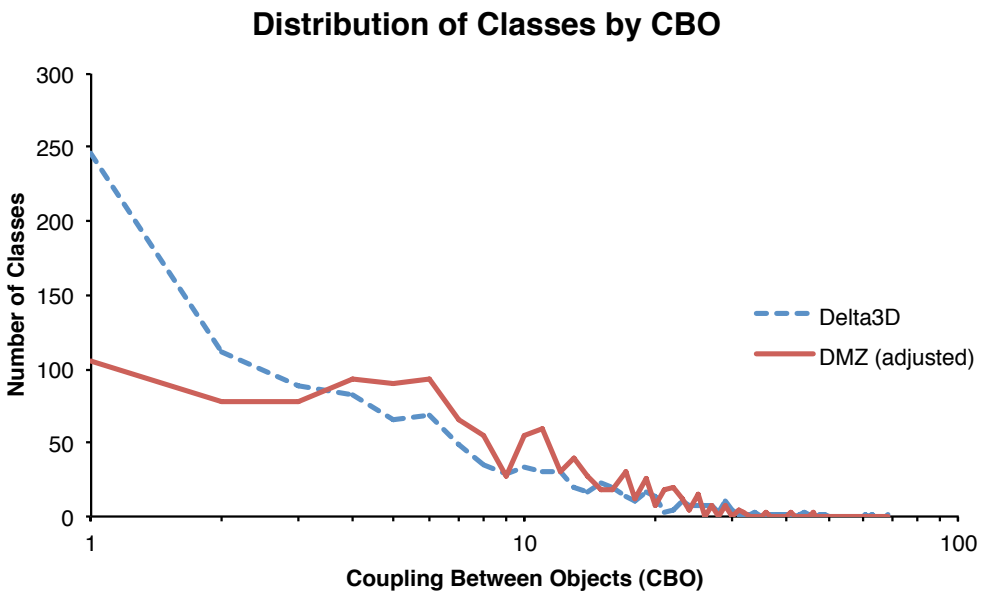


Figure 14: Number of classes per framework plotted by CBO reveals that Delta3D has a greater number of low-CBO classes than DMZ.

Table 23: The top ten classes with the highest CBO values reveals that Delta3D has more of the highest-CBO values than DMZ.

Rank	Class	Framework	Layer	CBO
1	dtHLAGM::HLAComponent	Delta3D	Middleware	69
2	dtGame::GameManager	Delta3D	Kernel	63
3	dtDAL::ActorPropertySerializer	Delta3D	Kernel	61
4	dtDAL::NamedGroupParameter	Delta3D	Kernel	49
5	dtDirector::DirectorEditor	Delta3D	Middleware	48
6	dmz::ObjectModuleBasic	DMZ	Middleware	46
7	dtDirector::DirectorEditor	Delta3D	Middleware	46
8	dtActors::WaterGridActor	Delta3D	Middleware	46
9	dtCore::StatsHandler	Delta3D	Kernel	45
10	dtGUI::GUI	Delta3D	Middleware	44

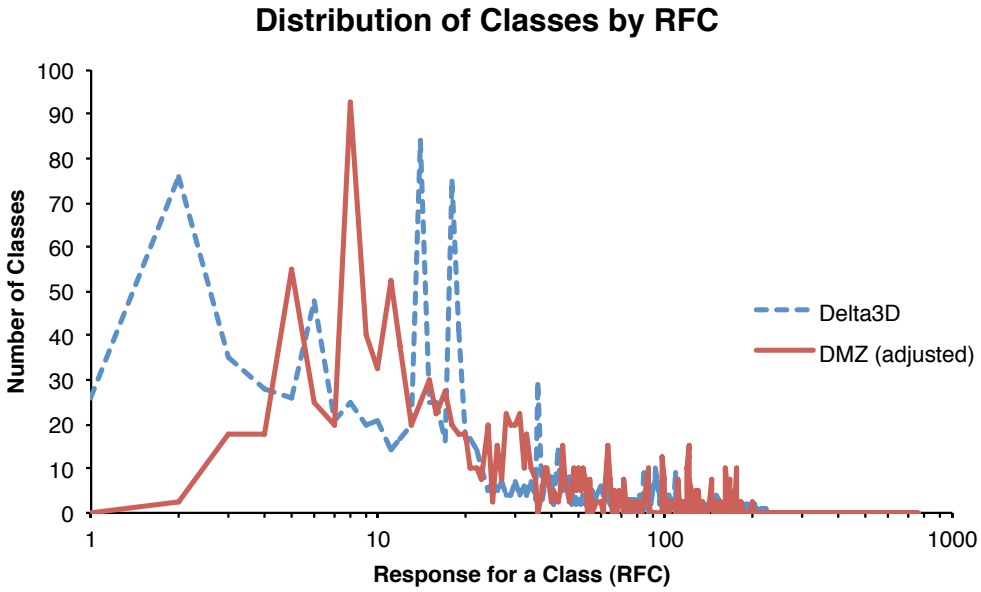


Figure 15: Number of classes per framework plotted by RFC reveals Delta3D with some small classes but otherwise beat by DMZ.

Table 24: The top eleven classes with the highest RFC values reveals that Delta3D has some of the highest-RFC classes.

Rank	Class	Framework	Layer	RFC
1	dtCore::RefPtr	Delta3D	Kernel	852
2	dtActors::WeatherEnvironmentActor	Delta3D	Middleware	224
3	dtActors::WaterGridActor	Delta3D	Middleware	219
4	dtActors::TaskActorGameEvent	Delta3D	Middleware	217
5	dtActors::TaskActorOrdered	Delta3D	Middleware	210
6	dtActors::TaskActorRollup	Delta3D	Middleware	207
7	dtActors::TaskActor	Delta3D	Middleware	205
8	dtActors::SkyDomeEnvironmentActor	Delta3D	Middleware	204
9	dtActors::CoordinateConfigActor	Delta3D	Middleware	204
10	dtActors::DirectorActor	Delta3D	Middleware	203
11	dmz::ObjectModuleBasic	DMZ	Middleware	202

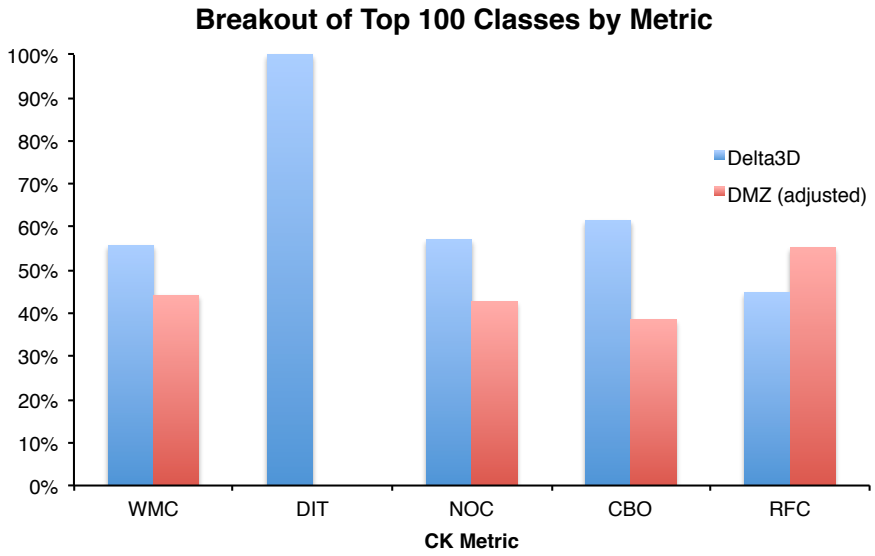


Figure 16: DMZ has fewer classes in the top 100 (except RFC), sorted descending by metric, than Delta3D, even adjusting for DMZ being 60% smaller than Delta3D.

Table 25: DMZ has fewer classes in the top 100 (except RFC), sorted descending by metric, than Delta3D, even adjusting for DMZ being 60% smaller than Delta3D.

Framework	WMC	DIT	NOC	CBO	RFC
Delta3D	55.8	100.0	57.2	61.5	44.7
DMZ	44.2	0.0	42.8	38.5	55.3

C. SUMMARY

We have successfully shown that our reuse model differentiates between two visual simulation frameworks and that we gain valuable insight in the process of applying and interpreting the model. The model contributes a new approach and tool for program managers and others to assess the nature of visual simulation frameworks with respect to the potential for reuse.

The model does not present as dramatic a difference between the frameworks as was expected. The proponents of component architecture tout the focus on decoupling, but the model seemed to focus more on the reusability of the underlying objects on which the components were built than the components themselves. For that, new metrics of agility are developed in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

V. AGILITY

In this chapter we develop and apply a model for assessing agility based on the definition and need established in Chapter II: Literature Review. The model uses new metrics that were created out of necessity for lack of any established metrics. We conduct a study in which we show that the model differentiates between two visual simulation frameworks and dramatically illustrates the differences in the two architectures.

A. DEVELOPING THE AGILITY MODEL

We learned that "'agility is a very seductive word" and that there are many "personal definitions" of it (Dove, 1994), but we require a model that is more precise. Measuring agility as we define it, by code being easily reconfigured, repurposed, or integrated, is not as straightforward as the static analysis we used for openness or reuse. We have defined agility in terms of actions—reconfigure, repurpose, integrate—and so agility must be measured "in flight" as actions take place.

1. Measuring Agility

Agility is sometimes linked to component based programming, and there has been some effort in developing complexity and other metrics specifically for components (Sharma & Kumar, 2007; Ismail, Wan-Kadir, Saman, & Mohd-Hashim, 2008; Qingqing & Xinke, 2009). These metrics try to do for components what CK metrics do for objects and classes, but they do not address our specific question of agility.

We could find no literature offering metrics for measuring software agility, but Lanman and Proctor (2009) remind us of the value of swapping out components in a system. Therefore, swapping out functionality in software is where we will turn. This action touches on reconfiguring, repurposing, and integration.

Being able to swap out components is a useful thing. It is easy for developers to think that once a piece of code is working, they will never go back to it, although this could

be due to nothing more than burn out from working with the same code for a long time. Whatever the reason code inevitably needs to be replaced or upgraded, because requirements inevitably change. Developers should be planning on change as the only constant, and something that improves the change process is worth considering.

It is not feasible to measure directly the effort required to swap out a portion of a simulation framework. Effort could be measured in time or money. It would be wasteful of both to go through the whole process of swapping out code just to see how hard it was. Instead we require metrics that will estimate the effort that *would* be required to execute the swap.

a. Included Files

We can estimate the effort of swapping out a piece of code by counting the connections to those files that are being replaced. Many programming languages have an `include` or `import` statement that makes available the functionality of an external piece of code. The more code we have that connects to this swapped out code, the greater the effort required to complete the swap. We therefore propose four metrics that measure the extent to which files are included that will need to be removed as part of the swapping process (Table 26).

Functionality F is being swapped out. L is the set of all lines of code in the project. C is the set of all classes in the project. I_F is the set of all `include` statements for functionality F. L_{I_F} is the set of all lines of code that `include` functionality F. C_{I_F} is the set of all classes that `include` functionality F. $|X|$ is the cardinality of set X.

Table 26: Agility metrics related to the files that are included.

Metric	Formulation
Lines that Include F (LI)	$LI = \|L_{I_F}\|$
Percent of Lines that Include F (PLI)	$PLI = \|L_{I_F}\| \div \|L\|$
Classes that Include F (CI)	$CI = \|C_{I_F}\|$
Percent of Classes that Include F (PCI)	$PCI = \|C_{I_F}\| \div \|C\|$

It is possible that some files may have `#include` statements but do not actually use functionality F in the file. While such files should be easier to fix than others, they still must be inspected and altered for the removal of functionality F to be complete, and so the metrics retain their usefulness even in this situation.

b. References

Counting the `#include` statements is a good start and provides a good, coarse metric for estimating the effort required for swapping out functionality, but there are at least two issues that it leaves unaddressed: the case of unnecessary, "left over" `#include` statements abandoned in the code and the *amount* of code within each file that actually uses the functionality. To address these, additional metrics that count the actual references to elements, *e.g.*, functions or variables, from functionality F are added (Table 27).

R_F is the set of all operations that use functionality F. L_{R_F} is the set of all lines of code that use one ore more operations in R_F. C_{R_F} is the set of all classes that use one ore more operations in R_F.

Table 27: Agility metrics related to references made.

Metric	Formulation
Lines with References to F (LR)	$LR = \lvert L_{R_F} \rvert$
Percent of Lines with References to F (PLR)	$PLR = \lvert L_{R_F} \rvert \div \lvert L \rvert$
Classes with References to F (CR)	$CR = \lvert C_{R_F} \rvert$
Percent of Classes with References to F (PCR)	$PCR = \lvert C_{R_F} \rvert \div \lvert C \rvert$

2. Model for Assessing Agility

To assess agility we select a functionality and estimate the effort to swap it out. The eight agility metrics are calculated, but the actual process of swapping out the functionality is not completed as that would entail unnecessary cost.

Unlike the openness and reuse models we have no information on which to create criteria against which to measure these metrics objectively. Instead multiple frameworks can be compared by comparing the metrics (Table 28).

Table 28: Agility metrics can be compared side by side.

	Framework	
Metric	Framework 1	Framework 2
Lines of Code, |L|	xx,xxx	xx,xxx
Number of Classes, |C|	...	...
Lines with Includes, LI		
Percent of Lines with Includes, PLI		
Classes with Includes, CI		
Percent of Classes with Includes, PCI		
Lines with References, LR		
Percent of Lines with References, PLR		
Classes with References, CR		
Percent of Classes with References, PCR		

B. STUDY 3: ASSESSING AGILITY

To demonstrate the feasibility of this assessment model, two simulation frameworks Delta3D and DMZ (described in Sections 2 and 3) are analyzed with the agility model by estimating the effort required to swap out the rendering engines. The rendering engine is perhaps the most complex portion of a visual simulation framework, so performing well in this test is a significant challenge. This is a worst case scenario. In both frameworks, the rendering engine is OSG. This helps provide a clean environment for verifying the usefulness of the model.

1. Methodology

To calculate the metrics in Tables 26 and 27, scripts (Appendix B) were run against the source code in each framework.

After the counting metrics are calculated, the percentages are calculated based on the number of lines of code or number of classes, as appropriate, and as given by the tool *Understand* used earlier in this research.

The data was then compiled in spreadsheets to be aggregated by framework, layer, or other division. Calculations, charts, and analysis was conducted from within these spreadsheets.

2. Results

The experiment is quick to execute, although the effort represented by the metrics is still considerable. A more thorough experiment, though of questionable additional value, would be to fund the teams of developers to implement this change and record the time and money that was required. Table 29 shows the results of the calculations.

Table 29: The agility model suggests almost an order of magnitude improvement of DMZ over Delta3D.

	Framework	
Metric	Delta3D	DMZ
Lines of Code, |L|	160,705	98,336
Number of Classes, |C|	1,191	475
Lines with Includes, LI	935	158
Percent of Lines with Includes, PLI	0.6	0.2
Classes with Includes, CI	310	31
Percent of Classes with Includes, PCI	26.0	6.5
Lines with References, LR	4,687	680
Percent of Lines with References, PLR	2.9	0.7
Classes with References, CR	259	31
Percent of Classes with References, PCR	21.7	6.5

a. Dramatic Differences

The results are also presented graphically in Figure 17. A striking differentiation can be seen in all four charts. The upper left chart, for example, reads, "Regard-

ing includes, Delta3D had 935 OSG includes, and DMZ had 158. Regarding references, Delta3D had 4,687 references, and DMZ had 680." This is almost an order of magnitude difference. It represents actual time programmers would need to spend pulling out Open Scene Graph and putting in something else. Removing size from the equation, the lower right chart is also compelling. It reads, "Regarding includes, 26.0% of Delta3D classes had OSG includes, and 6% of DMZ classes had OSG includes. Regarding references, 22% of Delta3D classes had OSG references, and 6% of DMZ classes had OSG references." This is a significant difference.

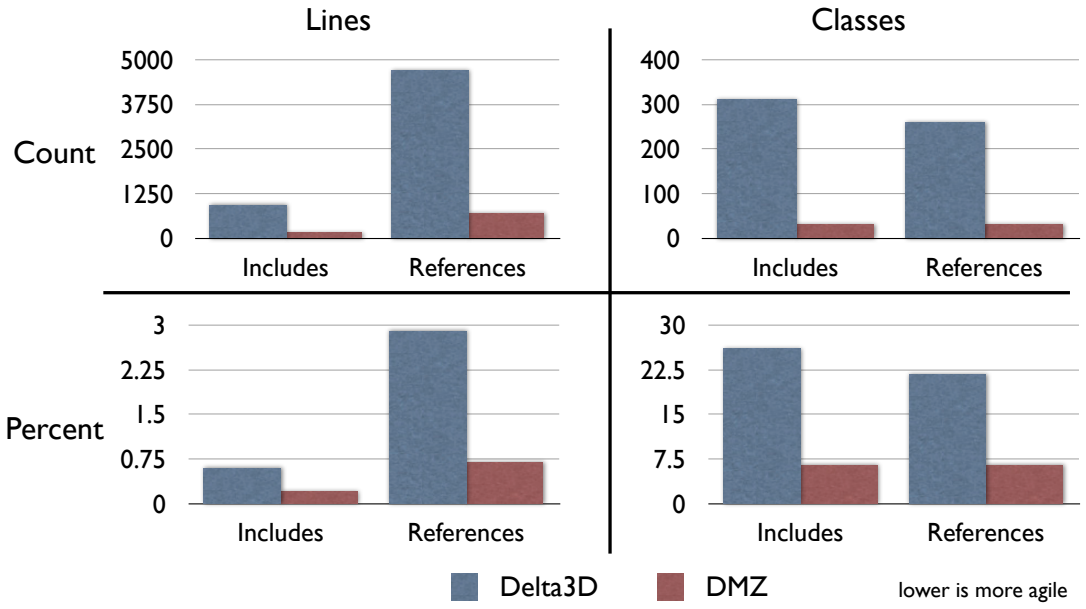


Figure 17: Whether by lines of code, by class, by raw count, or by percentage, the component based DMZ reveals itself to be more agile than Delta3D.

Yet another way to look at the results is with a Kiviat diagram (Figure 18), normalized with the Delta3D scores around the perimeter. This again shows a dramatic difference between the two frameworks when measuring agility.

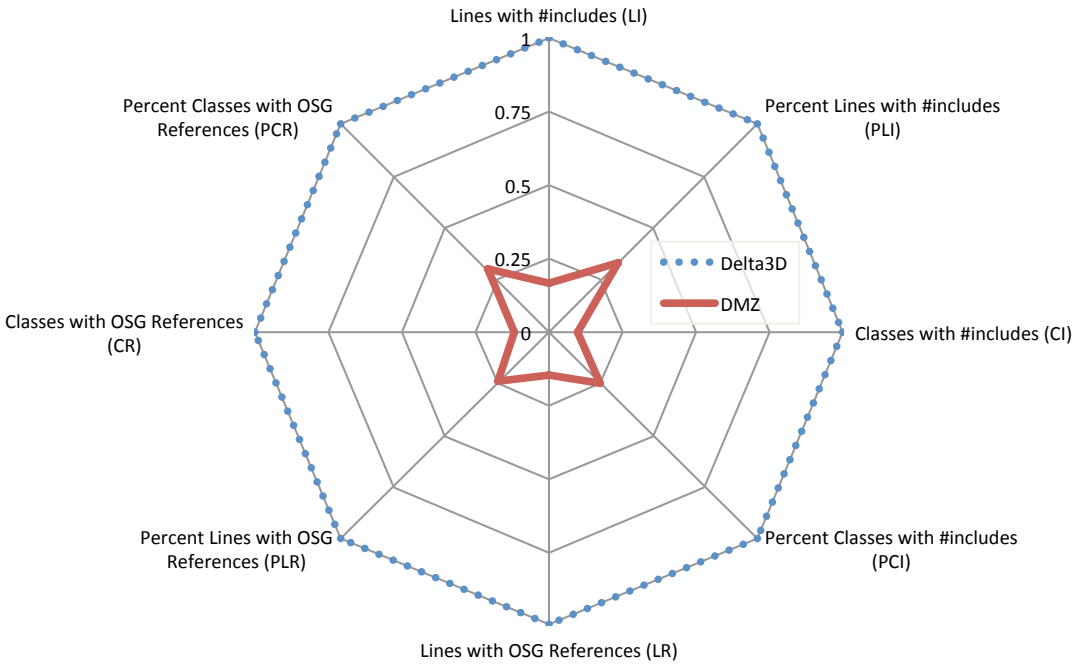


Figure 18: The estimated effort to swap out the rendering engine of DMZ (inner polygon, as a percent of Delta3D) is considerably less than Delta3D.

b. Includes vs. References

We might expect to see some correlation between some of the "files included" and "references" metrics, and that seems to be borne out. For example, DMZ has 31 classes that have OSG `include` statements and 31 classes that make OSG references, but breaking out the metrics this way also reveals some inconsistencies in Delta3D. There are 310 classes that have OSG `include` statements but only 259 classes with OSG references. This suggests "left over" code in Delta3D that could be cleaned up. A future study with more frameworks might determine if the *include* and *reference* metrics could be consolidated.

c. Absolutes vs. Percents

The metrics which provide absolute counts and the metrics with percentages offer different insight and should both be retained. The absolute counts, *e.g.*, LR and CR, are a doorway to estimating the time and money that would be required to make the changes represented in the study. The percentages provide a convenient, normalized metric that is more appropriate for assessing the quality of the code independent of its size.

C. SUMMARY

We have successfully shown that our agility model differentiates between two visual simulation frameworks and that we gain valuable insight in the process of applying and interpreting the model. The model contributes a new approach and tool for program managers and others to assess the nature of visual simulation frameworks with respect to openness.

VI. CONCLUSION

A. REVIEW

Although there is much guidance in the literature and in government documents for acquisition professionals regarding openness, reuse, and agility, there are no quantitative models that a PM could use to compare visual simulation architectures. This research contributed three models that provide quantitative differentiation for openness, reuse, and agility.

For each objective, a different amount of help was available from the literature (Table 30). With openness, the literature had taxonomies and terms, but this research developed the metrics and rolled them into a useful model. With reuse, the literature had definitions and even metrics, but this research brought the metrics together into a coherent model. With agility, the literature provided very little foundation, except to establish that people wanted "agility." This research provided a working definition, developed the metrics, and made some sense of the metrics all from scratch. For each objective this research included a study to demonstrate feasibility, and those were successful.

Table 30: For each objective, there was a different amount of help available from the literature.

	Openness	Reuse	Agility
Define terms	*Lit.*	*Lit.*	•
Develop metrics	•	*Lit.*	•
Build quantitative model	•	•	•
Perform feasibility study	•	•	•

B. DISCUSSION

1. Choice of Models

Prior to this research, there were few methodologies available to quantitatively identify the advantages of one architecture over another. The comparisons of one architecture to another seemed to hinge more on "brochure-ware" than scientific analysis. There did not seem to be a rigorous testing method for assessing attributes of simulation software that people supposed were important.

Openness and reuse were identified as candidate attributes. There is a lot of demand in the government for these qualities. However, these qualities have not been rigorously defined. There was much literature addressing various meanings of openness and various attributes related to reuse. There were definitions for the terms and some attempts at quantifying them, but there was nothing available to assess visual simulation software architectures.

The openness model developed in this research differentiated between the two frameworks studied. Both frameworks had similar open source licenses. Therefore, the strongest differentiation was made regarding standards and innovation. The model revealed significant differences in the consistency across the layers and the development operations. These differences would not be revealed by a licensing-only approach to openness.

The reuse model was the most surprising. Component based architectures stress decoupling. This suggested that the CK metrics that relate to coupling would reveal this strength of components over objects. This was not so. The component architecture was not a clear winner. Although the two frameworks were significantly different in their architecture, the reuse model showed only marginal differences.

It is the composability of component architectures that is attractive. Although both objects and components are meant to be reusable and both encourage programming to clear interfaces, components go a step further and abstract even the communication mechanisms. Components try to reduce coupling not by eliminating coupling itself but by changing the kind of coupling that happens. Objects can be though of as puzzle pieces—clearly

defined, well-documented, interchangeable even—but puzzle pieces nonetheless. Consider components to be Tinkertoys with a variety of shapes but very few ways to connect them (Figure 19).

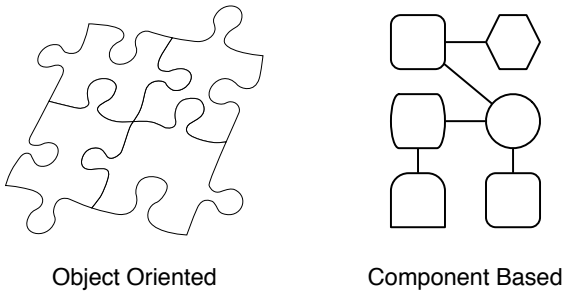


Figure 19: Objects can be likened to puzzle pieces that fit neatly together, and components can be likened to Tinkertoys with their various shapes connected by simple means.

This realization moved the research in a new direction toward agility. Measuring agility required measurements to be taken "in motion" rather than through static code analysis. This produced the methodology of swapping out a portion of a framework's functionality and estimating the effort of this action. The difference between the two frameworks studied was significant.

2. Additional Agility Metric

Some alternate analysis for assessing agility was introduced too late in the research for in depth study, but a brief summary is presented here. The central question revolves around how much variation is seen in the OSG references that are made within the frameworks. The idea is that a developer would have an easier time swapping out OSG from a class that had 100 references to the same OSG class, than a class with 1 reference each to 100 different OSG classes.

To investigate whether or not this could differentiate the two frameworks, the agility study was augmented to include a new metric, Response Entropy for a Class (REC). This metric is similar to the RFC CK metric, in that it represents a kind of "damage control" estimate. It is also similar in measurement technique to CBO. To calculate REC, for each

class in the framework that makes references to the functionality being swapped out, count the number of different classes from that functionality that are referenced. For example, if the OSG class `osg::Vec3` was referenced 20 times within a class, and that was the only class referenced from OSG, then REC for that class would be 1. The REC values were calculated for Delta3D and DMZ and aggregated by layer.

Table 31 shows the REC values for the classes that had OSG references. For comparison, the CBO values are also presented both for the classes that had OSG references as well as for all the classes in the framework. Also of interest is the observation that the CBO average values for the classes with OSG references was much lower than the averages for the entire frameworks.

Table 31: Agility REC values compared to CBO values for Delta3D and DMZ

			CBO	
Code	Classes	REC	These Classes	All Classes
Delta3D	259	8.2	15.3	7.43
Middleware	167	6.0	15.5	7.32
Kernel	92	12.3	15.1	7.69
DMZ	31	10.8	17.6	8.37
Middleware	31	10.8	17.6	10.24
Kernel	-	-	-	4.54

The REC appears to provide differentiation. It also provides an interesting way to reason about the effort required for swapping out parts. As such this metric deserves further exploration in future work.

The correlation between REC and CBO (Figure 20) is expected. Although REC is related to RFC for intent, it is also related to CBO in actual measurement. An increase in the REC necessarily implies an increase in CBO.

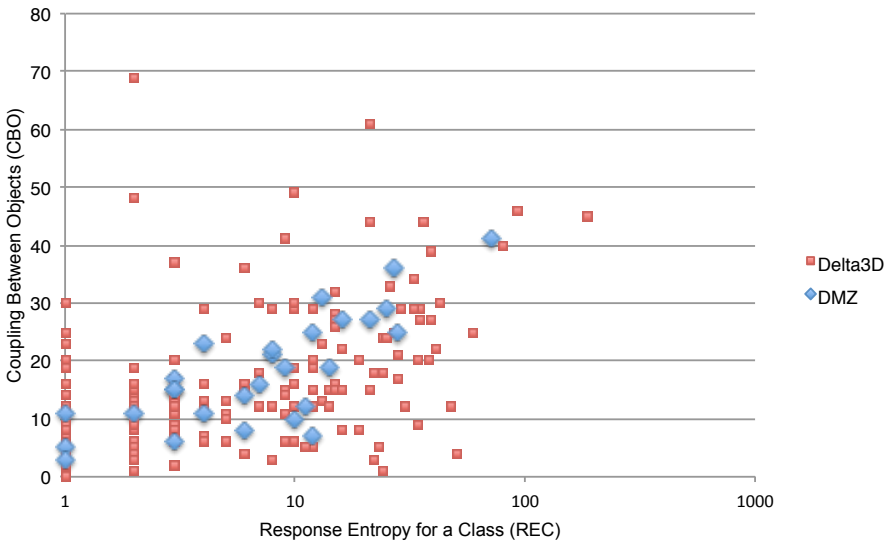


Figure 20: As expected, REC correlates to CBO

3. Uniformity of Design

An unexpected discovery during the openness study (Section B) was that the component framework DMZ had more consistent ratings across software layers and development operations than the conventional object based Delta3D. Recall that with regard to standards Delta3D scored I_s, II_s, and III_s for integrating, extending, and modifying the middleware, while DMZ scored a consistent I_s for all three because all interactions with DMZ revolve around configuring its plugins and manipulating data in its Object Module.

The value of this consistency is hard to determine. Because the integrate operation is the primary means by which a programmer uses a framework, this consistency may not be important to some developers. However, that view might be short-sighted. Today a framework may do just what the programmer needs it to do, but tomorrow it may fall short. An architecture with a consistent means for performing all development operations may help protect against the lock-in often seen today. Simply being *able* to perform all development operations is a win, but consistency therein is even better.

Consider the case of Virtual Battlespace 2 (VBS2), the "video game" from Bohemia Interactive developed for the military. VBS2 has an Application Scripting Interface (ASI) that allows developers to add features and behaviors. Evertsz and Ritter (2009) combine the cognitive architecture CoJACK with VBS2 to create suicide bomber scenario where fear and morale are modeled in the virtual actors. They remark that they are limited by what data and functionality is available in the scripting language. Not all elements of the simulation are accessible. Bohemia Interactive released VBS2Fusion which is advertised as protecting developers from having to learn their scripting language by letting them use C++ instead. It has expanded support for accessing data within the simulation (SimCentric, 2011). The impact of this new product, purchased separately from VBS2, has yet to be determined, but military customers are clearly held over a barrel as they try to integrate, extend, and modify their simulations.

The advantage of a uniform design may go deeper still. Extended applications (third party simulations) built with DMZ are subject to the same uniformity as the middleware. This means that, barring poor licensing negotiations with vendors, simulations built with an architecture like DMZ allow the government not only greater latitude in expanding the power of the framework but also greater ability to pick out and repurpose components from third party simulations. This may take us one step closer to avoiding the dreaded, "But we already paid for that!" complaint.

4. Helping the Three Program Managers

Chapter I introduced three PMs with three different needs for visual simulation as examples of stakeholders who could benefit from this research. One PM had a project with a horizon of 15 years or more. Another expected many other projects to fork off from the original. The third needed a quick and dirty simulation to answer a narrow and specific question. These three PMs might not weight the metrics the same and might eliminate certain elements of the model.

These three PMs were given weights for the openness and reuse models based on the things that were important to them. The models differentiated between the two frameworks, but with the weights applied, the model was able to present results from each PM's perspective.

Figure 21 shows that because of the way openness matters to the first two PMs, DMZ scores higher than Delta3D. For the PM who has an exercise coming up and just wants a quick analysis, openness is not as important, and the differences between the frameworks are marginalized.

With reuse the two frameworks were much more similar. Two things can be observed in the chart on the right. One is that two of the PMs care more about reuse than the third. This is similar to openness. Also the differences between the two frameworks are not great, since the difference in the ratings were not so dramatic with reuse.

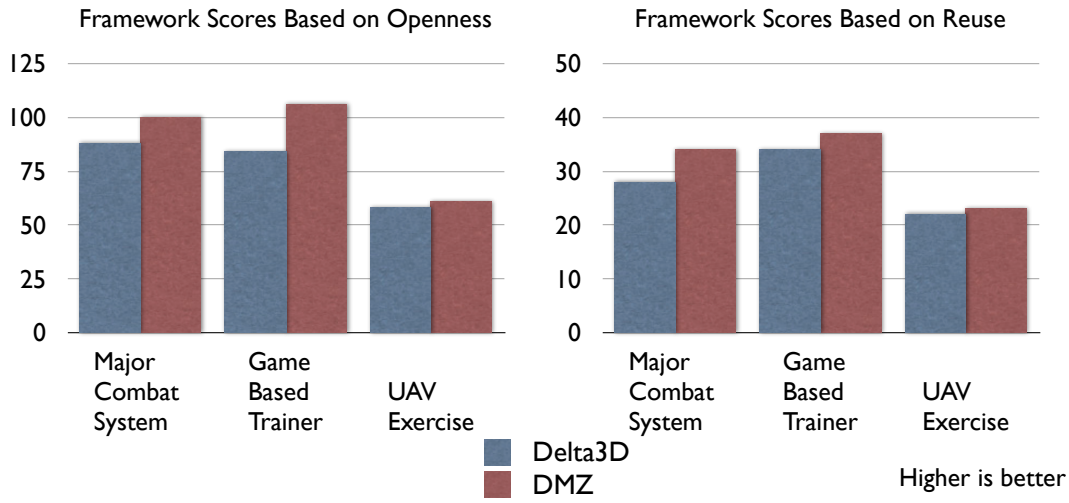


Figure 21: Different priorities among program managers may cause them to value openness, reuse, and agility—and hence different architectures—differently.

Agility is the least mature model, for obvious reasons. It has the least support from literature. The Kiviat diagram in Figure 18 is a compelling visual for PMs who are concerned about agility.

C. CONCLUSIONS

Openness, reuse, and agility are indeed important attributes of visual simulation frameworks, and they can be used to assess simulation frameworks and their underlying architectures. The three models developed in this research were successful at differentiating the two visual simulation frameworks studied and provided insight that could help acquisition professionals make decisions regarding their choice of simulation architectures.

In the literature we see that openness, reuse, and agility are important and that there are no suitable models for assessing visual simulation with respect to these qualities. Breaking openness into standards, licensing, and innovation on one axis (Maxwell, 2006) and software layers and development operations on two more axes (Anvaari & Jansen, 2010), provides a powerful space in which to characterize visual simulation, and possibly other, software. The venerable CK metrics (S. Chidamber & Kemerer, 1991) are well-tested and have many years of use to their credit. There is little to help build an agility model.

The three-axis openness model was a more powerful differentiator than was expected. It distinguished between two open source frameworks in the areas of open standards and open innovation. It revealed interesting features about one component based architecture regarding its uniformity of design (Section 3) that may help reduce the painful effects of legacy code in the future. Weighting the ratings allows for customizing an analysis of frameworks based on the particular needs of the project at hand.

The reuse model was the weakest differentiator of the three but did succeed in highlighting potential hot spots in the frameworks. In one case examining average metric values made one framework more appealing while examining the worst ranking classes made the other framework more appealing (Section a). The metrics related to coupling did not offer a convincing case for the advantages of components of objects, as was supposed. This effect encouraged the development of a third model based on agility (Section 1). Weighting the ratings allows for customizing an analysis of frameworks based on the particular needs of the project at hand.

The agility model was built without the aid of supporting literature but still revealed the most dramatic differences between the two frameworks studied. The estimated difference in effort between the two frameworks was almost an order of magnitude. Some of the advantages that proponents of component architectures like to promote regarding composability and flexibility were borne out in this study (Section 2).

D. FUTURE WORK

As successful as the three models of openness, reuse, and agility are, there are other models for these same important areas that could be developed and compared to the three in this dissertation. This is the first attempt to provide quantitative models for assessing visual simulation frameworks, and there may be room for improvement.

The metrics in the reuse model could be replaced with other software metrics linked to reusability. The CK metrics were selected because they have stood for a long time, are respected, and have been validated in the literature, but there are dozens of other metrics (Xenos et al., 2000; Kitchenham, 2010), largely untested except for the papers wherein they were introduced, and some of them may have some advantages. Metrics that may differentiate the models better along reuse should replace the metrics used here.

The strong differences in agility between the two frameworks suggest that there is potential to explore this further and develop more advanced and detailed models that break up agility into finer grains as we have done with the openness model. The characteristics that Qumer and Henderson-Sellers (2006b) uses—flexibility, speed, leanness, learning, and responsiveness—in measuring agility in companies may provide an alternative path for defining agility in software as well.

Other areas besides openness, reuse, and agility could be pursued. Maintainability and security are two very different characteristics at which people nod their head saying yes, they want more of it, and these are but two of the many desirable qualities of software that could benefit from clear definitions and assessment tools.

The final goal of this dissertation and related future work is to provide PMs with quantitative tools that they can use to aid in their decision making about visual simulation development. Rigorous models that lead to better architectures will help the DoD build or buy better software.

A. APPENDIX

A. SCRIPTS FOR CALCULATING REUSE METRICS

1. Parent Script

The following script fires off all the required scripts for generating metrics.

```
#!/bin/sh
# RunAll.sh

if [ $# -lt 1 ]; then
    echo "USAGE: $(basename $0) understand-database.udb"
    echo "Be sure also that you have generated reports in Understand:"
    echo "  Reports menu -> Generate Reports"
    echo "Probably best to run from the directory with the .udb file."
    exit
fi

UDB="$1"
UDBDIR=$(dirname "$UDB")
UDBBASE=$(basename -s .udb "$UDB")

# Convert the Understand Reports file to several CSV files
if [ -f "${UDBDIR}/${UDBBASE}.txt" ]; then
  ~/metrics/scripts/Convert-Understand-Report-To-CSV.sh "${UDBDIR}/${UDBBASE}.txt"
  echo
else
  echo "$(tput setaf 1)Missing report file.$(tput sgr0)"
  echo "  You need to generate reports in Understand:"
  echo "  Reports menu -> Generate Reports"
  echo "  The file should be named $(tput setaf 2)${UDBDIR}/${UDBBASE}.txt$(tput sgr0)"
  exit 1
fi


# Runs some custom metrics using the Understand Perl API
~/metrics/scripts/RunAllPerl.sh "$UDB"


# Merge the CSV files that are keyed off of classnames
echo "Merging csv files to $(tput setaf 3)${UDBDIR}/${UDBBASE}-Merged_Class_Metrics.csv$(tput sgr0)"
~/metrics/scripts/MergeCsv.sh \
  ${UDBDIR}/${UDBBASE}-Class{_OO,}_Metrics_Report.csv \
  ${UDBDIR}/${UDBBASE}-{cj_classmetrics,coupling,c_class_filenames,c_halstead}.csv \
  > ${UDBDIR}/${UDBBASE}-Merged_Class_Metrics.csv


#${UDBBASE}-{c_class_filenames,cj_classmetrics,coupling}.csv \
```

2. Converting Reports

The following script was used to convert the project reports generated by *Understand*. Some necessary metrics could be extracted from the reports with some processing.

```
#!/bin/sh
# Convert-Understand-Report-To-CSV.sh

if [ $# -lt 1 ]; then
    echo "USAGE: $(basename $0) understand-metrics-file"
    exit
fi

# From Understand, Reports -> Generate Reports, creates a file
# with (part of it) looking like what you see below. Run this
# converter to make the Class Metrics Report section tabular.
#
#                                Class Metrics Report
#=============================================================================
#Acceleration:
#   Lines                        36
#   Lines Blank                   5
#   Lines Code                   31
#   Lines Comment                 0
#   Average Lines                 7
#   Average Lines Comment         0
#   Average Complexity            1
#   Maximum Complexity            1
#   Ratio Comment/Code         0.00
#

function convert(){
SECTIONTOSUMMARIZE="$1"
INPUTFILE="$2"
INPUTDIR=$(dirname "$INPUTFILE")
INPUTBASE=$(basename -s .txt "$INPUTFILE")
THISSCRIPT="$0"
THISSCRIPTDIR=$(dirname "$THISSCRIPT")
AWK=awk

if [ $# == 3 ]; then
  if [ "$3" = "-" ]; then
    # To Standard Out
    $AWK -f "${THISSCRIPTDIR}/Understand-Report-to-CSV.awk" \
      SectionToSummarize="$SECTIONTOSUMMARIZE" "$INPUTFILE"
  else
    # To Named File
    OUTPUTFILE=$2
    $AWK -f "${THISSCRIPTDIR}/Understand-Report-to-CSV.awk" \
      SectionToSummarize="$SECTIONTOSUMMARIZE" "$INPUTFILE" > "$OUTPUTFILE"
    echo "Output $(( $(wc -l $OUTPUTFILE | awk '{print $1}') - 1 )) records to $OUTPUTFILE"
  fi
else
  # Generate a default file name
  OUTPUTFILE=${INPUTDIR}/${INPUTBASE}-$(echo $SECTIONTOSUMMARIZE | tr ' ' '_').csv
  $AWK -f "$(dirname "$0")/Understand-Report-to-CSV.awk" \
    SectionToSummarize="$SECTIONTOSUMMARIZE" "$INPUTFILE" > "$OUTPUTFILE"
  echo "Output $(( $(wc -l $OUTPUTFILE | awk '{print $1}') - 1 )) records to
$(tput setaf 2)$OUTPUTFILE$(tput sgr0)"
fi
```

```
}

# Do conversions
echo "Converting Understand reports to individual csv files..."
convert "Class Metrics Report" "$1"
convert "Class OO Metrics Report" "$1"
convert "Program Unit Complexity Report" $1
convert "File Average Metrics" "$1"
convert "File Metrics" "$1"
convert "Program Unit Complexity" "$1"
```

3. Running Perl Scripts

The software *Understand* has a Perl API for advanced analysis. Here are the files used.

```
#!/bin/sh
# RunAllPerl.sh

if [ $# != 1 ]; then
    echo "USAGE: $(basename $0) understand-database.udb"
    echo "  Creates a number of metrics files based on the Understand Database"
    exit
fi

UDB="$1"
THISSCRIPT="$0"
UDBBASE=$(basename -s .udb "$UDB")
UDBDIR=$(dirname "$UDB")
THISSCRIPTDIR=$(dirname "$THISSCRIPT")


PROC=1
echo "Detecting number of processors..." `sysctl -n hw.ncpu` && PROC=`sysctl -n hw.ncpu`
/bin/ls ${THISSCRIPTDIR}/{cj_classmetrics,coupling,c_class_filenames,c_halstead}.pl | \
xargs -n 1 -P "$PROC" "${THISSCRIPTDIR}/RunOnePerl.sh" "$UDB"

exit


# Execute all the Perl scripts in "scripts" folder as Understand Perl modules
#for script in ${THISSCRIPTDIR}/{cj_classmetrics,coupling,c_class_filenames,c_halstead}.pl; do
#   scriptBase=$(basename -s .pl "$script")
#   CSVFILE="${UDBDIR}/${UDBBASE}-${scriptBase}.csv"
#   echo "Processing $(tput setaf 2)$script$(tput sgr0) to $(tput setaf 1)$CSVFILE$(tput sgr0)"
#   uperl "$script" -db "$UDB" -csv "$CSVFILE" > /dev/null
#done
```

```
#!/bin/sh
# RunOnePerl.sh

if [ $# != 2 ]; then
    echo "USAGE: $(basename $0) understand-database.udb understand-perl-script.pl"
    echo "  Runs the Perl script for Understand and adds a -csv command line argument."
    echo "  Aids in using xargs to speed up processing."
    exit
fi

THISSCRIPT="$0"
UDB="$1"
```

```
PERLSCRIPT="$2"
UDBBASE=$(basename -s .udb "$UDB")
UDBDIR=$(dirname "$UDB")
PERLSCRIPTDIR=$(dirname "$PERLSCRIPT")

scriptBase=$(basename -s .pl "$PERLSCRIPT")
CSVFILE="${UDBDIR}/${UDBBASE}-${scriptBase}.csv"
echo "Processing $(tput setaf 1)$PERLSCRIPT$(tput sgr0) to $(tput setaf 2)$CSVFILE$(tput sgr0)"
uperl "$PERLSCRIPT" -db "$UDB" -csv "$CSVFILE" > /dev/null
```

4. Merge CSV

The following script was used to merge multiple CSV files.

```
#!/bin/sh
# MergeCsv.sh

if [ $# -lt 1 ]; then
    echo "USAGE: $(basename $0) file1.csv file2.csv ..."
    echo "Merges two or more CSV files with the first columns being common keys"
    exit
fi

function Merge () {

  CSV1=$1
  CSV2=$2

  awk -F, -v File1=$CSV1 -v File2=$CSV2 '
  BEGIN {
    # Read in the second file and record the header and the data
    # in an array that we can pull out later. We will be cross-
    # referencing the first columns as the key to unite the rows.
    NeedHeaders1 = 1;
    NeedHeaders2 = 1;
    F2KeysCount = 0;
    while ( (getline < File2) > 0 ) {              # Reads each row of the second file
      if( NeedHeaders2 ){                          # Looking for the first row here
        for( i = 2; i <= NF; i++ ){                # Loop through column titles
          Headers2[i] = $i;                        # Save the column titles
        }
        Headers2["count"] = NF - 1;                # Save how many data columns we have
        NeedHeaders2 = 0;                          # Mark that we are done with headers
      } else {
        f2data[$1,0] = $0;                         # Save the whole row
        for( i = 2; i <= NF; i++ ){                # Loop through the columns of data
          f2data[$1,i] = $i;                       # Save the data in a 2D array
        }
        F2Keys[$1] = $1;                           # Keep separate list of keys (first column)
                                                   # We delete keys from this as they are used
                                                   # to see if any keys appeared in the second
                                                   # file but not in the first.
      }
    } # end while
    OFS=","
  }
  {
    # Headers
    if( NeedHeaders1 ){                              # Looking for first row here
      printf( "%s", $0 );                            # Print the whole CSV line from file 1
```

```
      Headers1["count"] = NF - 1;                    # Store number of columns which we
                                                     #  may need if there were keys in the
                                                     #  second file that do not show up here.
      for( i = 2; i <= Headers2["count"]+1; i++ ){   # Loop through columns of file 2
        printf( ",%s", Headers2[i] );                # Append column titles from file 2
      }
      printf( "\n" );                                # End of row
      NeedHeaders1 = 0;                              # Done with column titles
    } else {

      # Data
      printf( "%s", $0 );                            # Print the whole CSV line from file 1
      for( i = 2; i <= Headers2["count"]+1; i++ ){   # Loop through columns of file 2
        printf( ",%s", f2data[$1,i] );               # Append data from file 2
        delete F2Keys[$1];                           # Remove from list of unprocessed file 2 keys
      }
      printf( "\n" );                                # End of row
    }
  }
  END {

    # Append keys that were only in file 2
    for( key in F2Keys ){                            # Loop through unprocessed file 2 keys
      printf( "%s", key );                           # Write the key to the first column
      for( i = 2; i <= Headers1["count"]+1; i++ ){   # Add blanks to all the columns for
        printf( "," );                               #  file1 that did not have this key.
      }
      for( i = 2; i <= Headers2["count"]+1; i++ ){   # Loop over columns in file 2
        printf( ",%s", f2data[key,i] );              # Append actual data from file 2
        delete F2Keys[key];                          # Remove from list of unprocessed keys
      }

    }
  }
  '  $CSV1
}


CSVTEMP1="$(mktemp -t MergeCsv)"
CSVTEMP2="$(mktemp -t MergeCsv)"
trap "rm -f '$CSVTEMP1' '$CSVTEMP2'" 0             # EXIT
trap "rm -f '$CSVTEMP1' '$CSVTEMP2'; exit 1" 2     # INT
trap "rm -f '$CSVTEMP1' '$CSVTEMP2'; exit 1" 1 15  # HUP TERM


# Merge all CSV files
while [ "$1" ]; do
  echo "Adding $(tput setaf 1)$1$(tput sgr0)" >&2
  if [ -s "$CSVTEMP1" ]; then
    #echo "Merging two CSV files"
    Merge "$CSVTEMP1" - < "$1" > "$CSVTEMP2";
  else
    #echo "Establishing first CSV file"
    cat "$1" > "$CSVTEMP1"
    cat "$1" > "$CSVTEMP2"
  fi
  cat "$CSVTEMP2" > "$CSVTEMP1"
  shift
done

cat "$CSVTEMP1"
```

B. SCRIPTS FOR CALCULATING AGILITY METRICS

The metrics can be calculated with command line tools such as grep and awk. There is some risk of counting a line that is in a block comment (*...*\), but single line comments (\\...) are excluded.

1. Lines with Includes, LI

```
# Delta3D:
grep -r osg merged-src-inc | grep '#include' | grep -v '//' | wc -l

# DMZ:
grep -r osg foundation frameworks kernel | grep '#include' | grep -v '//' | wc -l
```

2. Classes with Includes, CI

```
# Delta3D:
grep -r osg merged-src-inc | grep '#include' | awk -F: '{ print $1 }' | \
sed 's/\(.*\)\..*/\1/' | sort -u | grep -v '//' | wc -l

# DMZ:
grep -r osg foundation frameworks kernel | grep '#include' | \
awk -F: '{ print $1 }' | sed 's/\(.*\)\..*/\1/' | sort -u | grep -v '//' | wc -l
```

3. Lines with References, LR

```
# Delta3D:
grep -r --include=*.cpp 'osg[a-zA-Z]*::' merged-src-inc | grep -v '//' | wc -l

# DMZ:

grep -r --include=*.cpp 'osg[a-zA-Z]*::' foundation frameworks kernel | grep -v '//' | wc -l
```

4. Classes with References, CR

```
# Delta3D:
grep -r --include=*.cpp 'osg[a-zA-Z]*::' merged-src-inc | grep -v '//' | \
awk -F: '{ print $1 }' | sort -u | wc -l
```

```
# DMZ:
grep -r --include=*.cpp 'osg[a-zA-Z]*::' foundation frameworks kernel | \
grep -v '//' | awk -F: '{ print $1 }' | sort -u | wc -l
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Agile Alliance. (2011). *Agile alliance home page.* Available from http://www.agilealliance .org/

Agrawal, R., Bayardo, R. J., Gruhl, D., & Papadimitriou, S. (2002). Vinci: A service-oriented architecture for rapid development of web applications. *Computer Networks*, *39*(5), 523–539.

Allen, E., & Khoshgoftaar, T. (1999). Measuring coupling and cohesion: an information-theory approach. *Software Metrics Symposium, 1999. Proceedings. Sixth International*, 119–127.

Ambler, S. (1998). *A realistic look at object-oriented reuse.* Available from http://drdobbs .com/architecture-and-design/184415594

Anvaari, M., & Jansen, S. (2010). Evaluating architectural openness in mobile software platforms. In *Fourth european conference on software architecture* (pp. 85–92). New York, New York, USA: ACM.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, *22*(10), 751–761.

Beck, K., Cockburn, A., Jeffries, R., & Highsmith, J. (2001). *Manifesto for agile software development.* Available from http://agilemanifesto.org/

Becker, G. S. (1965). A theory of the allocation of time. *The Economic Journal*, *75*(299), pp. 493–517.

Benkler, Y. (2006). *The wealth of networks: How social production transforms markets and freedom*. Yale University Press.

Biggerstaff, T., & Richter, C. (1987). Reusability framework, assessment, and directions. *IEEE Software*, *4*(2), 41–49.

Briand, L. C., Daly, J., Porter, V., & Wust, J. (1998). A comprehensive empirical validation of design measures for object-oriented systems. *Software Metrics Symposium*, 246–257.

Briand, L. C., Daly, J., & Wust, J. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, *25*(1), 91–121.

Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, *22*(1), 68–86.

Brutzman, D., Zyda, M., Pullen, J. M., & Morse, K. L. (2002). Extensible modeling and simulation framework (xmsf) challenges for web-based modeling and simulation. *Strategic Opportunities Symposium*.

Cartwright, M., & Shepperd, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Trans. Softw. Eng.*, *26*(8), 786–796.

Champeaux, D. de, Lea, D., & Faure, P. (1992). The process of object-oriented design. *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*.

Chidamber, S., Darcy, D., & Kemerer, C. (1998). Managerial use of metrics for object-oriented software: an exploratory analysis. *Software Engineering, IEEE Transactions on*, *24*(8), 629–639.

Chidamber, S., & Kemerer, C. (1991). Towards a metrics suite for object oriented design. *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, *20*(6), 476–493.

Cho, E. S., Kim, M. S., & Kim, S. D. (2007). Component metrics to measure component quality. In *Eighth asia-pacific software engineering conference* (pp. 419–426). Macao, China: IEEE Comput. Soc.

Clinger, W., & Cohen, W. (1996). S.1124 national defense authorization act for fiscal year 1996 (Clinger-Cohen act).

Cohen, D., Lindvall, M., & Costa, P. (2004). An introduction to agile methods. *Advances in Computers*, *62*, 1–66.

Curfman, B. (1993). *Library development handbook. central archive for reusable defense software (cards)*. Electronic Systems Center Air Force Materiel Command.

Davis, P. K., & Anderson, R. H. (2004). Improving the composability of dod models and simulations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, *1*(1), 5–17.

Defense Acquisition University. (2010). *Defense acquisition guidebook*. Defense Acquisition University.

Department of Defense. (2003). *DoD directive 5000.01 the defense acquisition system*.

Department of Defense. (2008). *DoD instruction 5000.02 operation of the defense acquisition system*.

Department of Defense. (2011). *DoD open source software (oss) faq*. Available from http://cio-nii.defense.gov/sites/oss/Open_Source_Software_(OSS)_FAQ.htm

Devanbu, P., Brachman, R., & Selfridge, P. G. (1991). Lassie: a knowledge-based software information system. *Communications of the ACM*, *34*(5), 34–49.

Dove, R. (1994). *Agility essay #1: The meaning of life and the meaning of agility*. Available from http://www.parshift.com/Essays/essay001.htm

Economist. (2005). *Mashing the web*. Available from http://www.economist.com/node/4368150

Edwards, M. A. (2005). *Requirement for open architecture (OA) implementation*. Deputy Chief of Naval Operations Memorandum.

Evertsz, R., & Ritter, F. E. (2009). Populating VBS2 with realistic virtual actors. In *Proceedings of the 18th conference on behavior representation in modeling and simulation* (pp. 1–8). Sundance, UT.

Feig, N. (2008). *Pursuing the promise of SOA*. Available from http://www.banktech.com/management-strategies/208701020

Fong, G. (2004). Proceedings of the 2004 ACM SIGGRAPH international conference on virtual reality continuum and its applications in industry. In *Proceedings of the 2004 ACM SIGGRAPH international conference on virtual reality continuum and its applications in industry* (pp. 269–272). New York: ACM Press.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Garlan, D., Allen, R., & Ockerbloom, J. (2009). Architectural mismatch: Why reuse is still so hard. *Software, IEEE*, *26*(4), 66–69.

Gimenes, R., Silva, D. C., Reis, L. P., & Oliveira, E. (2008). Flight simulation environments applied to agent-based autonomous UAVs. In *International conference on enterprise information systems* (pp. 243–246).

Haefliger, S., Krogh, G. von, & Spaeth, S. (2008). Code reuse in open source software. *Management Science*, *54*(1), 180–193.

Halstead, M. H. (1977). *Elements of software science (operating and programming systems series)*. New York, NY, USA: Elsevier Science Ltd.

Henry, S., & Kafura, D. (1984). The evaluation of software systems' structure using quantitative software metrics. *Software: Practice and Experience*.

Herz, J. C., Lucas, M., & Scott, J. (2006). *Open technology development roadmap plan.* Available from http://www.acq.osd.mil/jctd/articles/OTDRoadmapFinal.pdf

Hippel, E. von. (2005). *Democratizing innovation*. Cambridge, MA: The MIT Press.

Hitz, M., & Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Proc. intl. sym. on applied corporate computing.*

Ismail, S., Wan-Kadir, W. M. N., Saman, Y. M., & Mohd-Hashim, S. Z. (2008). A review on the component evaluation approaches to support software reuse. In *2008 international symposium on information technology* (pp. 1–6). IEEE.

Jansen, S., Brinkkemper, S., Hunink, I., & Demir, C. (2008). Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE Software*, *25*(6), 42–49.

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (1986). Software complexity measurement. *Communications of the ACM*, *29*(11).

Kim, Y., & Stohr, E. (1992). Software reuse: issues and research directions. In *System sciences, 1992. proceedings of the twenty-fifth Hawaii international conference on* (pp. 612–623).

Kitchenham, B. (2010). What's up with software metrics? a preliminary mapping study. *Journal of Systems and Software*, *83*, 37–51.

Krygiel, A. J. (1999). *Behind the wizard's curtain: An integration environment for a system of systems*. Washington, D.C.: C4ISR Cooperative Research Program.

Lanman, J. T., & Proctor, M. D. (2009). Governance of data initialization for service oriented architecture-based military simulation and command and control federations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, *6*(1), 5–16.

Lewis, J. (2006). *A common component-based software architecture for military and commercial pc-based virtual simulation*. Unpublished doctoral dissertation, College of Engineering and Computer Science University of Central Florida.

Lewis, M., & Jacobson, J. (2002). Game engines in scientific research. *Communications of the ACM*, *45*, 27–31.

Li, W., & Henry, S. (1993a). Maintenance metrics for the object oriented paradigm. *Software Metrics Symposium, 1993. Proceedings., First International*, 52–60.

Li, W., & Henry, S. (1993b). Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, *23*(2), 111–122.

Lynn III, W. J. (2010). The pentagon's cyberstrategy. *foreignaffairs.com*.

Martin, R. (1994). OO design quality metrics. *Self Published*.

Maxwell, E. (2006). Open standards, open source, and open innovation: Harnessing the benefits of openness. *Innovations: Technology, Governance, Globalization*, *1*(3), 119–176.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, *SE-2*(4), 308–320.

McDowell, P., Darken, R., Sullivan, J., & Johnson, E. (2006). Delta3D: A complete open source game and simulation engine for building military training systems. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, *3*(3), 143–154.

McIlroy, M. D. (1968). Mass produced software components. *Proceedings of the NATO Software Conference*, 79–85.

Mili, H., Mili, F., & Mili, A. (1995). Reusing software: issues and research directions. *IEEE Transactions on Software Engineering*, *21*(6), 528–562.

Mills, E. E. (1988). *Software metrics SEI curriculum module SEI-CM-12-1.1*. Seattle University: Carnegie Mellon University Software Engineering Institute.

Morad, S., & Kuflik, T. (2005). Conventional and open source software reuse at orbotech – an industrial experience. *IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05)*, 110–117.

Müller, T. (2011). How to choose an free and open source integrated library system. *OCLC Systems & Services: International digital library perspectives*, *27*(1), 57–78.

National Research Council. (2005). *Modeling and simulation in manufacturing and defense systems acquisition*. National Academy of Sciences.

Office of the Under Secretary of Defense. (2002). *DoD 5000.2-R mandatory procedures for major acquisition defense programs (MDAPS) and major automated information systems (MAIS) acquisition programs*. U. S. Department of Commerce.

Office of the Under Secretary of Defense. (2006). *Department of defense acquisition modeling and simulation master plan.* Office of the Under Secretary of Defense (Acquisition, Technology and Logistcis).

Offutt, J., Abdurazik, A., & Schach, S. (2008). Quantitatively measuring object-oriented couplings. *Software Quality Control*, *16*(4).

Olague, H. M., Etzkorn, L. H., Gholston, S., & Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes.

IEEE Trans. Softw. Eng., *33*(6), 402–419.

Ommering, R. van. (2005). Software reuse in product populations. *IEEE Trans. Softw. Eng.*, *31*(7), 537–550.

Open Source Initiative OSI. (2011). *The open source definition.* Available from http://opensource.org/docs/osd

Open Systems Joint Task Force. (2004). Program manager's guide. *Open Systems Joint Task Force*.

OSGForum. (2011). *Openscenegraph forum – ive file format.* Available from http://forum.openscenegraph.org/viewtopic.php?t=7878

Oswalt, I. (1993). Current applications, trends, and organizations in U.S. military simulation and gaming. *Simulation & Gaming*, *24*(2), 153–189.

Poulin, J. S. (2006). The business case for software reuse: Reuse metrics, economic models, organizational issues, and case studies. *Lockheed Martin Distribution Technologies*.

Prather, R. E. (1984). An axiomatic theory of software complexity measure. *The Computer Journal*, *27*, 340–347.

Qingqing, Z., & Xinke, L. (2009). Complexity metrics for service-oriented systems. In *2009 second international symposium on knowledge acquisition and modeling* (pp. 375–378). IEEE.

Qumer, A., & Henderson-Sellers, B. (2006a). Comparative evaluation of XP and Scrum using the 4D analytical tool (4-DAT). In *European and mediterranean conference on information systems.* Costa Blanca, Spain.

Qumer, A., & Henderson-Sellers, B. (2006b). Measuring agility and adoptability of agile methods: A 4-dimensional analytical tool. *IADIS International Conference Applied Computing*, 503–507.

Ragab, S. R., & Ammar, H. H. (2010). Object oriented design metrics and tools: A survey. In *7th international conference on informatics and systems* (pp. 1–7). Cairo, Egypt.

Rosenberg, L. H., & Hyatt, L. E. (1995). *Software quality metrics for object-oriented environments: A report of SATC's research on OO metrics* (Tech. Rep.).

Schwartz, R., & Phipps, S. (2011). *Floss weekly podcast: Open source software at the department of defense.* Available from http://twit.tv/floss160

Scott, J. M. (2010). *Pentagon is losing the softwar(e).* Available from http://www.defensenews.com/story.php?i=4677662

Scott, J. M. (2011). *Open technology development (odt): Lessons learned and best practices for military software.* Office of the DoD CIO.

Sharble, R. C., & Cohen, S. S. (1993). The object-oriented brewery. *SIGSOFT Softw. Eng. Notes*, *18*(2), 60–73.

Sharma, A., & Kumar, R. (2007). Managing component-based systems with reusable components. *International Journal of Computer Science and Security*, *1*(2), 52–57.

SimCentric. (2011). *Overview: VBS2Fusion.* Available from http://www.simcentric.com .au/products/vbs2fusion/overview/

Stallman, R. (2010). *Why open source misses the point of free software.* Available from http://www.gnu.org/philosophy/open-source-misses-the-point.html

Tang, M.-H., Kao, M.-H., & Chen, M.-H. (1999). Proceedings sixth international software metrics symposium (cat. no.pr00403). In *Metrics '99: Sixth international symposium on software metrics* (pp. 242–249). IEEE Comput. Soc.

Tangney, J. (2009). Synthetic environments for assessment. *NPS Guest Lecture*.

TechWeb. (2008). *The state of service-oriented architecture.* Available from http://www .slideshare.net/techweb08/techweb-state-of-soa-research/download

Therriault, R. W., & Van Nederveen, K. E. (1994). *Industry versus DoD: A comparative study of software reuse*. Unpublished doctoral dissertation, Naval Postgraduate School, Monterey.

Tufte, E. R. (2001). *The visual display of quantitative information* (2nd ed.). Cheshire, CT: Graphics Press.

Updegrove, A. (2005). *Consortiuminfo.org consortium standards bulletin- march 2005.* Available from http://www.consortiuminfo.org/bulletins/mar05.php#feature

U.S. Air Force. (2004). *Air force instruction 33-114*. U.S. Air Force.

Vessey, I., & Weber, R. (1984). Research on structured programming: An empiricist's evaluation. *Software Engineering, IEEE Transactions on*, *SE-10*(4), 397–407.

Wand, Y., & Weber, R. (1990). An ontological model of an information system. *Software Engineering, IEEE Transactions on*, *16*(11), 1282–1292.

Washizaki, H., Yamamoto, H., & Fukazawa, Y. (2003). A metrics suite for measuring reusability of software components. *5th International Workshop on Enterprise Networking and Computing in Healthcare Industry*, 211–223.

WaughPartners, & OSSWatch. (2007). *The foundations of open.* Available from http:// pipka.org/blog/2008/07/23/the-foundations-of-openness/

Wennergren, D. M. (2009). *Clarifying guidance regarding open source software (OSS).* Department of Defense Chief Information Officer.

Weyuker, E. J. (1988). Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, *14*(9), 1357–1365.

White, I. (1994). *Using the booch method: A rational approach*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.

Wichmann, T. (2002). *Free/libre open source software: Survey and study* (Tech. Rep.). Berlecon Research.

Wilkes, M. V., & Renwick, W. (1949). The EDSAC – an electronic calculating machine. *Journal of Scientific Instruments and of Physics in Industry*, *26*, 385–391.

Xenos, M., Stavrinoudis, D., Zikouli, K., & Christodoulakis, D. (2000). Object-oriented metrics – a survey. In *Federation of european software measurement association 2000* (pp. 1–10). Madrid.

Xu, T., Qian, K., & He, X. (2006). Service oriented dynamic decoupling metrics. In *2006 international conference on semantic web and web services* (pp. 1–7). Las Vegas.

Yilmaz, L. (2004). On the need for contextualized introspective models to improve reuse and composability of defense simulations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, *1*(3), 141–151.

Young Jr., J. J. (2004). *Naval open architecture scope and responsibilities*. Assistant Secretary of the Navy Memorandum.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Col. David Gibson
   Department of Computer Science
   U.S. Air Force Academy
   Colorado Springs, Colorado

4. Dr. Ken Doerr
   Graduate School of Business and Public Policy
   Naval Postgraduate School
   Monterey, California

5. Dr. John Tangney
   ONR341 Division Director

6. LCDR Joseph Cohn, PhD
   ONR341 Deputy Division Director

7. James O'Bryon
   The O'Bryon Group
   Bel Air, Maryland

8. Dr. Tom West
   Oregon State University
   Corvallis, Oregon